Пояснювальна записка
до дипломного проекту
на тему:
«Система контролю здорового способу життя»

Київ – 2019 рік

ЗМІСТ

СПИСОК ВИКОРИСТАНИХ СКОРОЧЕНЬ

API – Application Programming Interface;

HTML – HyperText Markup Language;

HTTP – HyperText Transfer Protocol;

IP – Internet Protocol;

JSON – JavaScript Object Notation;

JRE – Java Runtime Environment;

JWT – Json Web Token;

ORM – Object-Relational Mapping;

REST – Representational State Transfer;

SOAP – Simple Object Access Protocol;

SQL – Structured Query Language;

UML – Unified Modeling Language;

XML – eXtensible Markup Language;

YAML – YAML Ain't Markup Language

ОС – операційна система;

СУБД – система управління базами даних

ВСТУП

На сьогоднішній день на ринку існує багато фітнес-застосунків для різних платформ. Абсолютна більшість з них пропонує готові комплекси тренувань, харчування за додаткову сплату. Проте, дуже мало додатків, що дозволяють користувачеві самостійно скласти собі комплекс будь-яких вправ, що задовільняють його спосіб життя. Також, вони не враховують, що кожен день користувача може бути розписаний по хвилинах і про заплановані фізичні активності він може просто забути, чи йому може не вистачити мотивації та психічних сил. Існуючі додатки, якщо й здатні відправляти нагадування, то лише у вигляді вспливаючого вікна на мобільному пристрої при увімкненій мобільній передачі даних.

Проектована система покликана вирішити ці проблеми. Користувач зможе абсолютно безкоштовно самостійно складати для себе гнучку програму тренувань, відстежувати свій прогрес набору чи втрати ваги та встановлювати нагадування зручними для нього способами – пропонується лист на електрону пошту, SMS-повідомлення чи телефонний дзвінок, при чому можна обрати декілька засобів одночасно. Вочевидь, окрім клієнтських апплікацій, така система мусить реалізовувати стійкий та гнучкий клієнт-серверний додаток.

Для реалізації цих потреб були вжиті наступні заходи:

- аналіз існуючих застосунків, спрямованих на контроль здоров’я;
- збір вимог до розробляємого продукту;
- ознайомлення з різними підходами до архітектури та вибір оптимального;
- вибір засобів розробки та супроводжувальних інструментів;
- реалізація серверної частини відповідного програмного забезпечення.

Проект побудовано за допомогою мови програмування Java, фреймворку для мікросервісної архітектури Spring Cloud, його під-компонентів, із застосу-

ванням SQL-бази даних PostgreSQL, NoSQL-бази MongoDB, кешоподібного сховища вигляду ключ-значення Redis, системи контейнеризації Docker та інструменту для автоматичного розгортання багатокомпонентних систем Docker-Compose.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Застосунок для смарт-браслету Notify & Fitness for Mi Band

Android-застосунок, який надає інформацію про фази сну, серцебиття, пройдені кроки, масу тіла через ручний смарт-браслет Xiaomi Mi Band. Дозволяє конфігурувати повідомлення та дзвінки на стороні браслета.

Переваги:

- вичерпна інформація про звичайні аспекти біологічної життєдіяльності: серцевий ритм, кількість кроків пройдених за день, трекінг калорійних витрат, індекс маси тіла;
- висока інтеграція із звичними Google-продуктами;
- широка кастомізація повідомлень.

Недоліки:

- немає можливості складати та відслідковувати програму тренувань;
- необхідність придбати смарт-браслет;
- містить багато реклами;
- багато функціоналу доступно лише за платну підписку.

1.2 Android-застосунок BetterMe: Weight Loss Workouts

Android-застосунок для контролю фітнес-тренувань та програми харчування, орієнтований на жіночу аудиторію.

Переваги:

- непогані запропоновані набори вправ;
- пропонується варіативний раціон здорового харчування;
- локалізація інтерфейсу англійською, іспанською, португальською та німецькою мовами;

– відсутність реклами.

Недоліки:

– нагадування та мотивуючі повідомлення лише через поп-ап на мобільному пристрої при ввімкненій мобільній передачі даних;
– орієнтація виключно на жіночу аудиторію;
– безкоштовне використання впродовж лише семи днів.

Висновки до розділу №1

Аналізуючи існуючі рішення на ринку, можна зробити висновок, що вони не задовільняють таких потреб, як:

– гнучкість у складанні програми тренувань;
– зручні нагадування;
– незалежність від сторонніх провайдерів аутентифікації;
– незалежність від станів застосунку, як-то явний запуск чи фоновий режим;
– незалежність від додаткових пристроїв зчитування показників стану організму;
– стійка, гнучка та легкомасштабована серверна частина;
– відсутність мікротранзакцій;
– відсутність рекламних блоків;
– орієнтація на широку аудиторію.

Тому проектування системи, яка б задовільняла цим потребам, має великий сенс.

2 ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ

Аналізуючи існуючі рішення, можна зробити висновок, що безкоштовного застосунку, що дозволяє користувачу повністю скласти для себе дуже гнучку програму тренувань та може впливати не тільки на власний процес мобільного додатку, зараз на ринку немає.

Існує думка, що люди, які користуються нагадувачами стосовно будь-яких своїх справ, на 60% успішніші за інших. Зовнішній мотиватор чи хоча б невелике ненав’язливе повідомлення може зіграти велику психологічну роль та майже стовідсотково перевернути рішення людини в сторону “Так”. Саме таким чином народилася ідея спортивного застосунку, що може подати сигнал користувачеві за межами своїх обов’язків через найпоширеніші сторонні сервіси, як-то лист на електронну пошту, SMS-повідомлення чи телефонний дзвінок.

Розроблювальна система орієнтована в першу чергу на мобільні застосунки. У них користувач зможе скласти гнучку програму тренувань, додавати, редагувати та видаляти тренування, різні види вправ та нагадувачі.

Також користувачеві буде надана можливість на інтерактивному екрані заповнити значення власної ваги до та після тренування, щоб у перспективі, побачити графік свого прогресу залежно від мети – набір чи скидання кілограмів. Більш повний опис використання застосунку можна побачити на діаграмі прецедентів Б.1.

Висновки до розділу №2

У проекті розглядається серверна частина системи. Оскільки вона мусить взаємодіяти з різними сховищами даних та сторонніми провайдерами прикладного програмного інтерфейсу, орієнтуватися на велику кількість користувачів, така система зобов’язана бути швидкою, стійкою, надійною, легкомасштабованою та легкопідтримуємою.

3 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

3.1 Монолітна архітектура

На сьогоднішний день на ринку домінують два підходи до побудови архітектури програмного забезпечення – монолітний та мікросервісний. Традиційний підхід до побудови веб-застосунків зображено на рисунку 3.1.

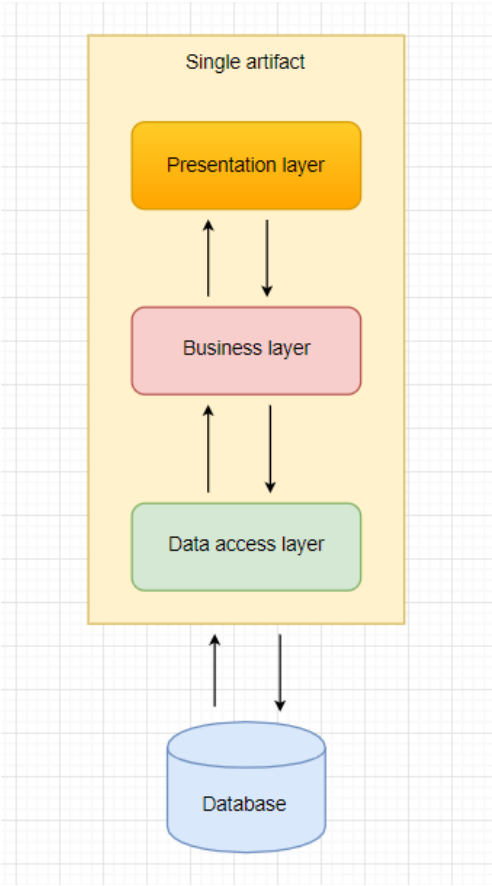

Рисунок 3.1 – Типова тришарова (монолітна) архітектура

В цьому підході серверна частина складається з трьох шарів – рівня презентації, рівня бізнес-логіки та рівня доступу до данних. Застосунки такого типу збираються в єдиний артифакт і його вже розгортають на сервері.

За декілька десятків років застосування принципу в ньому виявилося достатньо недоліків, серед яких:

– надмірна складність;
– зависока ціна комунікацій та координацій між командами;
– неможливість горизонтального масштабування;
– нестійкість та вразливість;
– жорстка прив’язаність до єдиного стеку технологій.

Під надмірною складністю мається на увазі, що бізнес-потреби зростають, відповідно, зростає і кількість нової кодової бази, що необхідно впровадити. В якийсь момент проект розростається до таких масштабів, що людський мозок не здатний охопити усі її внутрішні зв’язки та підтримувати таку систему стає майже неможливо.

Звичайно, сучасні системи контролю версій дозволяють легко керувати процесом розробки єдиного продукту між багатьма відділами. Однак, це не рятує від попереднього пункту. Кожного разу як одна з команд потребує впровадити зміну в код, увесь застосунок мусить заново пройти усі Unit-тести, збудуватися та розгорнутися. Із зростом складності програмного забезпечення розуміння продукту та комунікація між командами мусять бути на дуже високому рівні, а процеси перетестування, перебудови та перерозгортання починають вимірюватися годинами та добами.

Говорячи про проблеми масштабування, необхідно зазначити, що коли на якусь частину системи йде надмірне навантаження та відчутна проблема в продуктивності, операційний відділ замислюється над розгортання поруч іще декількох екземплярів системи, і встановленням попереду них балансувальника навантаження (load balancer). Проблема в тому, що не можна розширити лише той завантажений компонент, і доведеться розгортати іще як мінімум один повноцінний артифакт, який потребуватиме таких самих ресурсів, що й перший – частоти процесору, оперативної пам’яті, та ін. Доведеться додатково інвестувати у серверне обладнання.

Окрім того, оскільки застосунок збирається в єдиний артифакт, то в разі виникнення помилки в одній частині, «впаде» уся система, а також буде проблематично локалізувати цю помилку.

3.2 Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура прийшла на зміну монолітній у двохтисячних роках. Основна її ідея полягає в тому, що застосунок розподілений на декілька незалежних компонентів, які називаються сервісами. Ці сервіси можуть бути розподілені по різних серверах та спілкуватися один з одним за домогою технологічно-нейтрального протоколу: SOAP чи REST. Приклад такої архітектури зображено на рисунку 3.2.

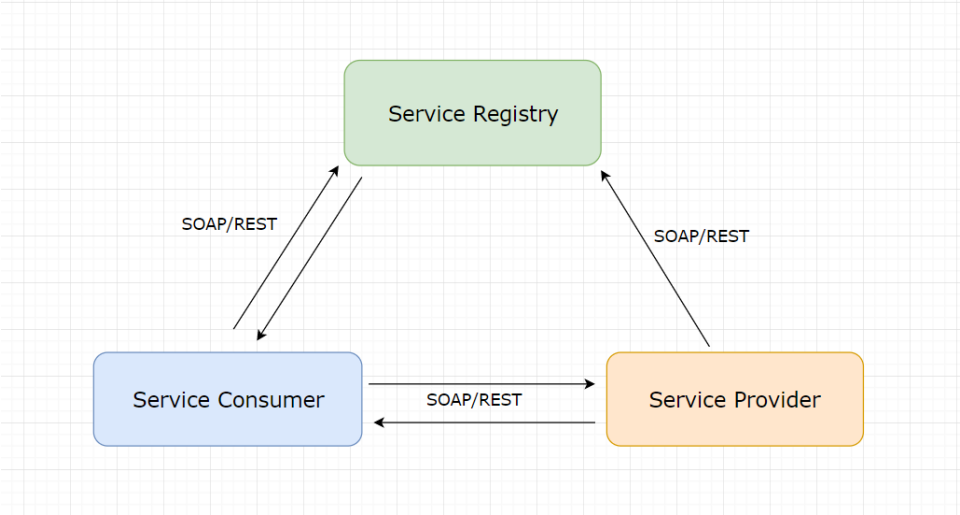

Рисунок 3.2 – Сервіс-орієнтована архітектура

У найпростішому сценарії в системі присутні три компоненти: реєстр сервісів, споживач сервісу та постачальник сервісу. Постачальник сервісу реєструє свої сервіси в реєстрі, а споживач звертається до реєстру, оскільки він не знає про місцезнаходження постачальника. До сервісів висуваються такі вимоги, як

стандартизованість інтерфейсів, відсутність стану, слабка зв’язність, перевикористання та автономність. Сервіс-орієнтована архітектура перед монолітною має наступні переваги:

- часткове розгортання;
- часткове масштабування;
- підвищена стійкість.

Кожен сервіс може бути розгорнутий незалежно від інших, відповідно, і масштабувати його можна окремо. Також, якщо станеться помилка в одному компоненті, решта система не постраждає. Тим не менш, ці сервіси все одно залишаються достатньо великими та ресурсомісткими.

Серед недоліків сервіс-орієнтованої архітектури можна виділити:

- все ще надмірна складність;
- жорстко-стандартизований протокол SOAP.

Не дивлячись на те, що сталась певна декомпозиція, сервіси все одно приймають на себе забагато відповідальності та схильні сильно розростатися. Також в ті часи методологія REST ще не набула популярності, та більшість подібних застосунків використовували Simple Object Access Protocol, обмінюючись даними у форматі XML. Інтерфейс кожного сервісу повинен був бути чітко визначеним, тому сервісам бракувало гнучкості .

3.3 Мікросервісна архітектура

Мікросервісний підхід до проектування архітектури народився у дві тисячі дванадцятому році та розвинувся з ідей сервіс-орієнтованої архітектури. Його зображено на рисунку 3.3.

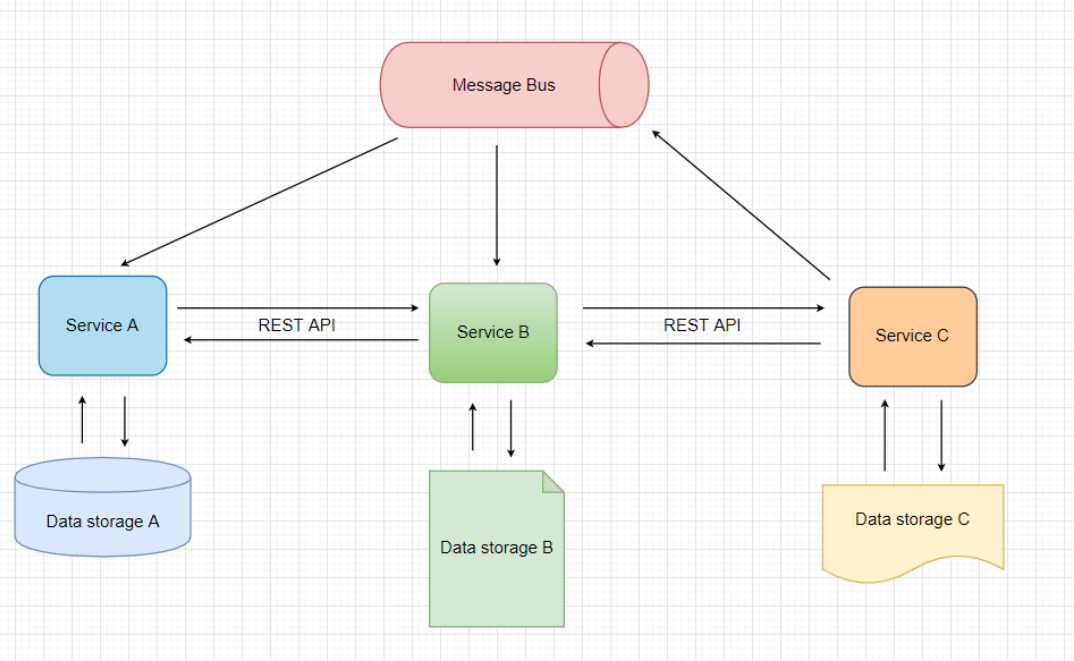

Рисунок 3.3 – Схематичне зображення мікросервісної архітектури та можливих засобів комунікації

У мікросервісному середовищі система декомпозується на декілька бізнес-сервісів, кожен з яких відповідає за лише одну доменну область, невеликий за кодовою базою, повністю незалежний від інших компонентів системи, може бути реалізований будь-якими мовами програмування та фреймворками та, як правило, має доступ до свого, відокремленого сховища даних, який найкраще підходить для його бізнес-задачі (підхід відомий як Polyglot Persistence). Сервіси можуть спілкуватися один з одним за допомогою будь-яких протоколів, що не залежать від обраних технологій.

Найрозповсюдженішими з них є REST API (синхронні виклики по HTTP) та обмін асинхронними повідомленнями через брокера.

У порівнянні з монолітною та сервіс-орієнтованою архітектурою, мікросервісна має наступні переваги:

– висока швидкість поставки нового функціоналу клієнтам;
– слабка зв’язність сервісів один з одним;
– простота підтримки кожного окремого сервісу;

– стійкість;
– горизонтальне масштабування;
– незалежність від вибору технологічного стеку.

Оскільки кожен сервіс невеликий за обсягом та не залежить від інших компонентів, кожна команда може швидко розробляти, тестувати, розгортати та постачати новий функціонал замовнику. Кожен сервіс відповідальний за одну частину бізнес-моделі та розгортається незалежно від інших, що дозволяє, у разі потреби, підняти більше екземплярів саме цього сервісу, а не всієї системи. Це не тільки спрощує масштабування, але ще й заощаджує серверні ресурси.

Надзвичайно важливий той факт, що сервіс не розростається у неконтрольованих масштабах, тому розробники не витрачають надмірні зусилля на його дослідження та підтримку.

У випадку «падіння» сервісу, решта системи не постраждає та проблему буде легко локалізувати. Сервіси спілкуються за допомогою технологічно-нейтральних протоколів, наприклад SOAP із даними у форматі виключно XML, чи REST із найпоширенішим форматом JSON. Це означає, що за потреби кожен сервіс може бути реалізований будь-якими засобами, що пасують до його конкретної бізнес-задачі.

Проте мікросервісна архітектура вимагає набагато більшого, ніж просто написання коду. Тепер розробникам програмного забезпечення доводиться вирішувати наступні проблеми:

– правильний розмір;
– незалежність від локації;
– балансування навантаження;
– повторюваність;
– стійкість;
– ізольованість;
– управління даними та розподілені транзакції;

– тестування, особливо міжсервісних сценаріїв;
– легка та швидка поставка змін;
– спостереження.

Бізнес-домен треба декомпозувати таким чином, щоб сервіси мали невеликий розмір та не брали на себе забагато відповідальності.

У мікросервісному застосунку багато екземплярів одного сервісу можуть бути швидко підняті та знищені, отже необхідно легко контролювати фізичну адресу для викликів сервісу, а також рівномірно розподіляти навантаження між екземплярами. Також кожен новий екземпляр сервісу зобов'язаний бути з тією самою конфігурацією та кодовою базою, що і існуючі екземпляри сервісу.

У випадку падіння сервісу, його споживачів та загальну цілісність систему необхідно захистити та забезпечити споживачам сервісу альтернативне джерело даних, або, хоча б, перекрити небезпечний шлях.

Бізнес-сервіси слід інкапсулювати від зовнішнього світу, щоб клієнт не мав до них прямого доступу. Логи та метрики з усіх компонентів розподіленої системи необхідно збирати в одне централізоване середовище щоб у разі проблем швидко прийняти міри.

Висновки до розділу №3

Очевидно, що система має бути гнучка, стійка, легкомасштабована та легкопідтримуєма. Тому мікросервісний підхід до проектування архітектури був обраний без жодних сумнівів.

4 ВИБІР ТА ОБГРУНТУВАННЯ ЗАСОБІВ РОЗРОБКИ

Ми живемо у епоху неймовірно стрімкого розвитку технологій. Інструменти для розробки, що з’являються на ринку, встигають застаріти швидше, ніж інженери їх опановують. Мікросервісна архітектура народилась у дві тисячі дванадцятому році. За цей час вона встигла неймовірно розвинутись та довести, що інвестиції ресурсів, що вкладаються у розробку на початку, сповна окупаються під час зросту продукту та бізнесу, що його задає. Увесь світ почав розробляти інформаційні системі на базі мікросервісної архітектури. Відповідно, з’являлося і продовжує з’являтися все більше і більше інструментів для вирішення тих чи інших прикладних задач. Далі розглядаються дві найпопулярніші мови програмування для backend-застосунків – Java та C# та дві найпопулярніші платформи від кожної з них – Spring Cloud та .NET Core відповідно.

Обидві мови на ринку вже близько двадцяти років, об’єктно-орієнтовані, строго типізовані, підтримують узагальнені типи, багатопотоковість та елементи функціонального програмування. C#, як мова, пішла у своєму розвитку далі за Java. Вона підтримує структури, делегати, події, має інтегровану SQL-подібну мову для запитів у будь-які сховища даних та неймовірно зручний синтаксис для контролю асинхронних операцій. Із виходом платформи .NET Core, яка прийшла на зміну старому .NET, backend-застосунки, написані мовою C# стали швидкими та кросплатформенними.

Проте, коли мова йде про веб-розробку, зрозуміло, що одними мовними засобами не обійдеться. Для клієнт-серверного застосунку необхідна система збірки, сторонні бібліотеки для взаємодії зі сховищами даних, серіалізацією об’єктів, способами комунікації, хмарними сервісами та сторонніми API. Оскільки Java тривалий час на ринку була стандартом та єдиним стабільним інструментом, під неї було створено безліч сторонніх бібліотек на будь-який випадок. Такого різноманіття бракує .NET, а тим паче молодому .NET Core.

В попередньому розділі розглядалися інфраструктурні виклики, які постають перед розробниками, які обрали мікросервісний підхід до проектування системи. Особливу увагу варто звернути на управління конфігураціями та та реєстрацію сервісів.

Java-фремворк Spring Cloud складається з підмодулів, що дозволяють надзвичайно швидно та просто вирішити багато з цих викликів. Netflix Eureka прямо з коробки дозволяє створити реєстр сервісів за допомогою лишень однієї анотації та конфігураційного файлу, а Spring Cloud Config дозволяє зберігати налаштування для сервісів окремо від них, при чому його можна підняти й локально.

У екосистемі .NET Core ці задачі можуть бути реалізовані лише через хмарні сервіси Microsoft Azure, та за допомогою контейнерного оркестратора Kubernetes, що не дозволяє розгорнути та протестувати систему локально.

Висновки до розділу №4

В ролі основної мови програмування та платформи для реалізації були обрані Java та фреймворк Spring Cloud. Як мова програмування, Java вже давно показала себе як стабільна, перевірена роками мова, що чудово справляється з задачами веб-розробки за рахунок величезної кількості різноманітних бібліотек, що для неї написані. Spring Cloud, що складається з багатьох проектів компанії Netflix з відкритим доступом, вирішує за розробника багато інфраструктурних питань, що були згадані вище. Усі мікросервіси, як бізнес-орієнтовані, так і інфраструктурні, будуватимуться за допомогою фреймворку Spring Boot через надзвичайно легкий старт розробки веб-застосунку, який він забезпечує.

5 СЦЕНАРІЇ ВИКОРИСТАННЯ СИСТЕМИ

Хоча у роботі і розглядається лише серверна частина, система орієнтована в першу чергу на мобільні додатки, тому в перспективі уся взаємодія з користувачем відбуватиметься через відповідні Android- чи IOS-застосунки.

Діаграма прецедентів – це UML-діаграма, яка допомагає описати взаємодію системи із зовнішнім середовищем, у випадку розроблюваної системи – з користувачем. Діаграма є графом, що складається з множини акторів, множини варіантів використання (прецедентів), асоціацій між прецедентами та акторами, відношеннями між прецедентами та відношеннями серед акторів. Її зображено на діаграмі ІА51.250БАК.005 Д1.

При першому запуску додатку користувач зможе створити новий аккаунт, чи виконати логін у існуючий. Якщо цей аккаунт вже був зареєстрований в системі, із серверу завантажиться існуюча програма тренувань, в іншому випадку – користувач зможе створити нові тренування.

На кожен день тижня користувач може додати тренування, вказати запланований час. У діалоговому вікні тренування він зможе додати вправи та інформацію про них, як-то назва вправи, робоча вага, тривалість виконання, тривалість відпочинку, кількість підходів та інше. Після створення тренування буде можливість його видалити, а також редагувати час та вправи для нього.

На інтерактивному екрані мобільного застосунку користувач зможе вводити свої показники по вправам, щоб слідкувати, чи виконує він власний план на цей тиждень, а також задавати власну вагу до та після тренуваня. В кінці тижня, його результати будуть збережені в локальне сховище на пристрої. Це зроблено для того, щоб сгенерувати за бажанням користувача графік зміни ваги за вказаний ним період.

Для кожного дня тренування користувачу буде надано можливість створити одне чи більше нагадувань різними засобами. Розроблювана система пропонує лист на електронну пошту, SMS-повідомлення та телефонні дзвінки. Усі створені нагадування можна буде редагувати чи видаляти.

Оскільки система орієнтована на мобільні пристрої, розробляти браузерний користувацький інтерфейс немає сенсу, особливо враховуючи статистику популярності мобільних застосунків у порівнянні з веб-сайтами.

Натомість, користувацький інтерфейс як Android-, так і IOS-додатків, має бути максимально приємний та інтуїтивно зрозумілий користувачеві.

Висновки до розділу №5

Функціонал, що буде реалізований, описаний в достатній мірі та не містить зайвих можливостей. Це дозволить у майбутньому зробити інтерфейс клієнтського застосунку простим та інтуїтивно-зрозумілим, а це означає – привабливим. В свою чергу, серверну частину системи можна буде легко розроблювати та горизонтально масштабувати, оскільки у її компонентів будуть чітко виділені межі.

6 ГОЛОВНІ ШАБЛОНИ ПРИ ПРОЕКТУВАННІ МІКРОСЕРВІСНИХ СИСТЕМ

6.1 Взаємодія з пристроями користувача

Для обміну данними з клієнтськими застосунками в даній системі був застосований паттерн «Єдина точка доступу» (також відомий як API Gateway). Єдина точка доступу – це шаблон проектування у розподілених системах, що відділяє виклики з клієнтських застосунків від внутрішніх сервісів. Подібну взаємодію зображено на рисунку 5.1.

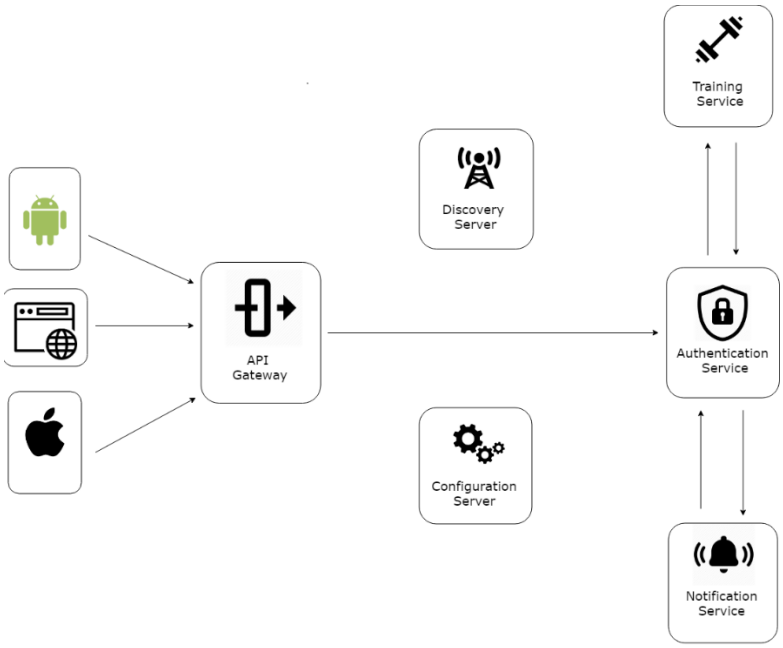

Рисунок 6.1 – Будь-які виклики до внутрішніх сервісів пройдуть лише через єдину точку входу

Такий підхід має багато переваг:

- інкапсуляція внутрішніх сервісів;
- маршрутизація;
- додатковий рівень захисту;
- фільтр дозволених клієнтів;

– інтеграція з Service Discovery та балансування навантаження;
– участь у зборі метрик та логів.

У зовнішній світ відкрито єдиний порт, і клієнти завжди абстраговані від фізичного місцезнаходження сервісів, що їх цікавлять. Єдина точка доступу надає можливість розумної маршрутизації до API внутрішніх сервісів, а також статичних IP-адрес чи доменних імен.

Гарною практикою також є реалізація на рівні Gateway логіки з авторизації та аутентифікації користувачів, що надає додатковий шар захисту для внутрішніх сервісів. Нові протоколи захищеної передачі інформації, такі як OAuth2, дозволяють скоротити набір клієнтських застосунків до декількох відомих, наприклад веб-браузер, Android чи IOS-застосунки, а також заборонити виклики, що йдуть зі специфічних IP-адрес.

У деяких програмних продуктах, що є реалізацією API Gateway, є можливість інтеграції із засобами реєстрації сервісів, що дозволяє проводити балансування навантаження між екземплярами сервісів.

Коли приходить запит на єдину точку входу, йому можна надати унікальний ідентифікатор – так званий Correlation Id, що, у разі помилки системи, допоможе відслідкувати усі етапи користувацької активності, через які пройшов цей запит.

6.2 Ізоляція клієнтів від фізичної локації сервісів та масштабування

Із зростом користувацької бази, бізнес-потреб та кількості коду у веб-застосунках рано чи пізно постає питання масштабування. Надзвичайно важливо враховувати майбутні проблеми та ризики ще на етапі проектування системи, особливо – мікросервісної. Коли сервіси вже набудуть якоїсь кодової масси, яка дозволить випустити застосунок на бойові сервери, якщо способи масштабування не були опрацювані заздалегідь – тепер вони коштуватимуть надто дорого.

Для цієї задачі існує два підходи: балансувальник навантаження (“Load balancer”) та реєстр сервісів (“Service discovery”). Балансувальник навантаження – це, зазвичай, готове програмне забезпечення від стороннього вендора, яке дозволяє розподіляти HTTP-виклики поміж статичної кількості серверів. З іншого боку, реєстр сервісів – критично важливий шаблон при проектуванні розподілених систем, особливо тих, для яких планується розгортання у хмарних середовищах. По-перше, він дозволяє швидко горизонтально масштабувати сервіси. Оскільки споживачі сервісів завжди абстраговані від їх фізичного місцезнаходження, операційний відділ може легко та швидко збільшувати та зменшувати кількість запущених екземплярів сервісу, не завдаючи шкоди клієнтам. По-друге, більшість реалізацій Service Discovery збільшують стійкість усієї системи шляхом автоматичного видалення «хворих» екземплярів з набору. Таким чином, клієнти будуть захищені від випадкових викликів до сервісів, що не відповідають.

Порівняння традиційного підходу із балансувальником навантаження та реєстром сервісів зображено на рисунках 6.2 та 6.3.

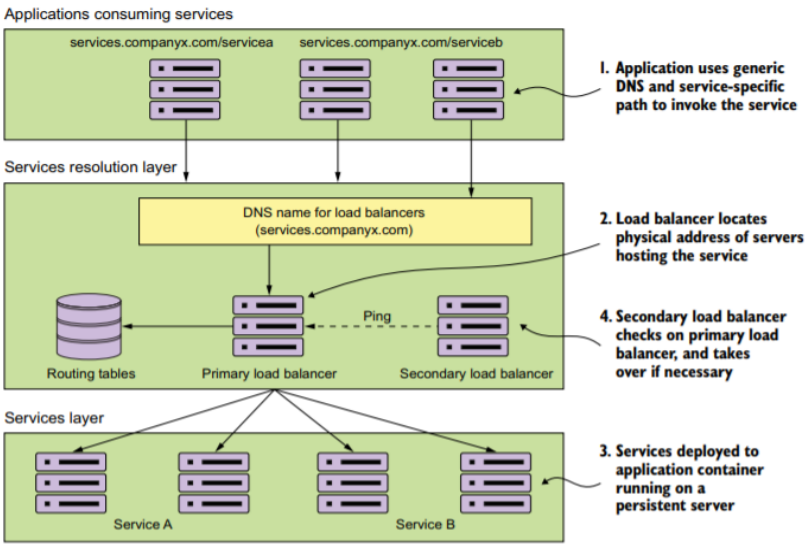

Рисунок 6.2 – Вертикальне масштабування з балансувальником навантаження

(John Carnell – “Spring Microservices in action”)

Клієнтські застосунки звертаються до адреси балансувальника та специфічного для внутрішнього сервісу маршруту. Первинний балансувальник має свою таблицю маршрутизації, з якої зчитує місцезнаходження конкретних екземплярів сервісів та перенаправляє запити до них. В свою чергу, вторинний балансувальник періодично викликає первинний, щоб переконатись у його працездатності, та, у випадку відсутності відповідей, прийняти контроль над користувацькими запитами на себе.

Такий підхід добре личить внутрішньокорпоративній інфраструктурі, коли у середовищі працює мала та обмежена кількість екземплярів сервісів зі статичними адресами серверів, які під них виділили. Проте, він зовсім не підходить для хмарно-орієнтованих мікросервісних застосунків. Причини для цього наступні:

- єдина точка падіння;
- обмежене горизонтальне масштабування;
- статичне управління;
- складність.

Не дивлячись на те, що балансувальник навантаження можна зробити високодоступним, все одно це означає, що у разі проблем з балансувальником, не буде доступний жоден із внутрішніх сервісів. Із статичною кількістю серверів, інфраструктура обмежена обчислювальними ресурсами, і, відповідно – кількістю екземплярів сервісів, які можна розгорнути.

Більшість комерційних балансувальників навантажень не призначені для динамічної реєстрації та дереєстрації нових екземплярів сервісів, використовують єдине сховище даних (наприклад, таблицю маршрутизації) та єдиний вихід додати новий маршрут – через специфічний API постачальника балансувальника. Оскільки балансувальники перенаправляють клієнтські запити, споживачі сервісів повинні прив’язувати свої запити до IP-адрес фізичних серверів. Це додає рівень складності до інфраструктури, оскільки правила маршрутизації для сервісів повинні бути визначені та запущені у використання вручну.

Рішення для хмарно-орієнтованих застосунків полягає у підході Service Discovery, який:

- легкодоступний;
- ділиться інформацією про екземпляри сервісів поміж кожного вуза в discovery-кластері;
- динамічно балансує навантаження;
- стійкий;
- швидко реагує при виявленні «мертвого» екземпляру сервісу.

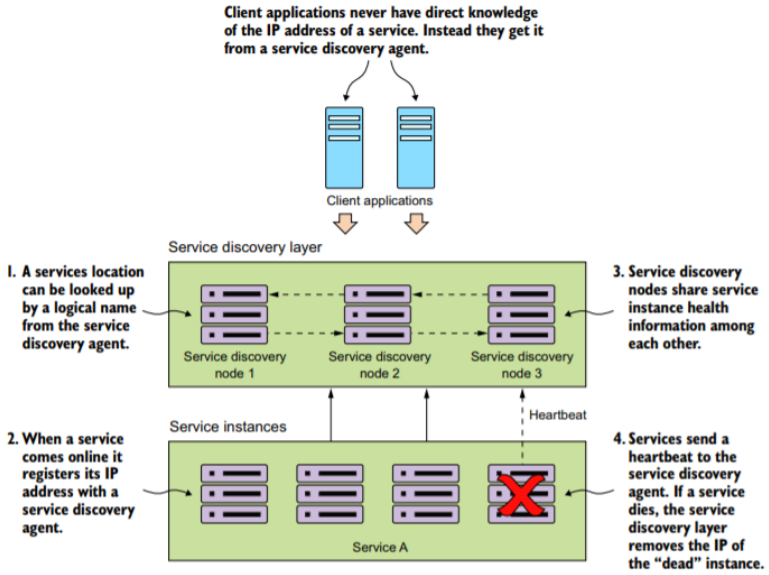

Рисунок 6.3 – Динамічна реєстрація екземплярів сервісу у реєстрі та горизонтальне масштабування
(John Carnell – "Spring Microservices in action")

Клієнтські застосунки ніколи не знають про фізичне місцезнаходження сервісів, в яких вони зацікавлені. Цю інформацію вони отримують від реєстра сервісів, який реєструє адресу та порт екземпляру сервісу за його логічним ім'ям. Кожен екземпляр реєстру сервісів поширює інформацію про місцезнаходження та стан екземплярів серед інших екземплярів реєстру в кластері.

Кожен екземпляр внутрішніх сервісів періодично відправляє так званий heartbeat – сигнал реєстру про свій стан. У випадку, якщо екземпляр перестав сигналізувати, реєстр видалить «мертвий» екземпляр зі своєї бази знань.

Проте, попри всі переваги, такий підхід також не є ідеальним – споживачі сервісів стають повністю залежними від service discovery механізму, який мусить бути працездатним, щоб знайти та викликати необхідний сервіс.

Більш надійним є підхід, що називається «клієнтське балансування» (Client-side load balancing). Рисунок 6.4 демонструє його.

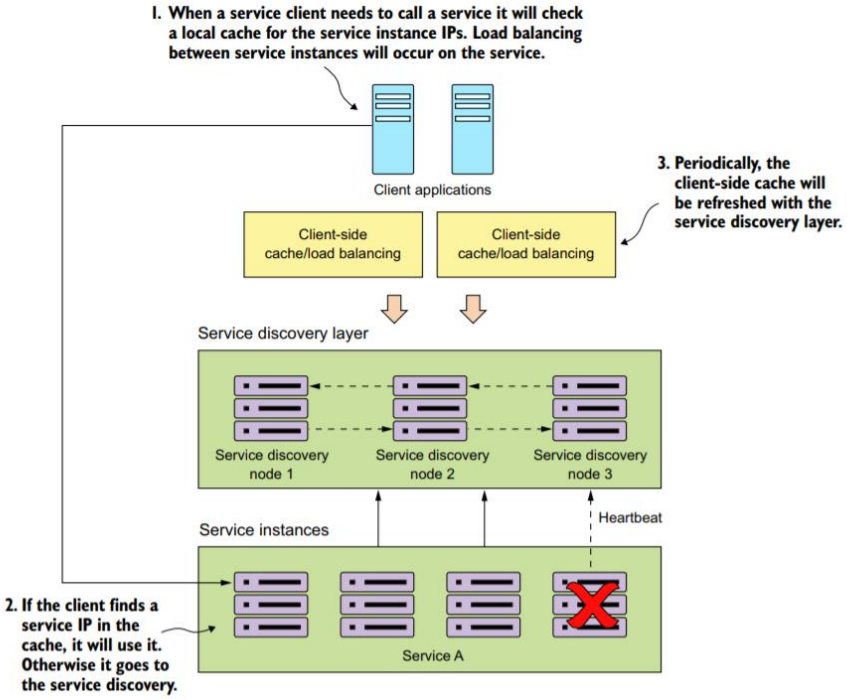

Рисунок 6.4 – Клієнтське балансування навантаження
(John Carnell – “Spring Microservices in action”)

Коли клієнтському застосунку вперше знадобиться викликати якийсь сервіс, він звернеться до реєстру сервісів, та збереже дані про місцезнаходження усіх екземплярів цього сервісу в своєму локальному кеші. Тепер щоразу, коли споживачу сервісу знадобиться якийсь сервіс, він буде шукати інформацію про

нього у себе в кеші. Зазвичай, кешування на стороні клієнта використовує простий алгоритм балансування, як, наприклад, «round-robin» з метою переконатися, що виклики до сервісу рівномірно розподілені серед усіх його екземплярів.

У разі, якщо за закешованою IP-адресою споживач не знайде сервіс, або не отримає від нього відповіді, він звернеться до реєстру сервісів з метою оновити адреси екземплярів у своєму локальному кеші. Також він здатний оновлювати свій кеш періодично, а не тільки за потреби.

В розроблюваній системі буде застосовано саме шаблон «клієнтське балансування» із застосуванням відповідної бібліотеки Ribbon від компанії Netflix, яка буде взаємодіяти із реєстром сервісів, реалізованим за допомогою Netflix Eureka.

6.3 Захист клієнтів від викликів до проблемних сервісів

Сучасні інформаційні системи не є малими та вузьконаправленними. Вони розширюються, обмінюються данними із іншими сервісами, постачальниками специфічної для бізнесу інформації від сторонніх вендорів та взаємодіють з різними видами сховищ даних. Частіше за все, подібний обмін данними відбувається за протоколом HTTP. Проте, команда програмістів та системних адміністраторів, що розробляє продукт, який інтегрується із сторонніми постачальниками інформації, не може бути впевнена, що ця інформація завжди буде доступна. Виклики до низькодоступних, “хворобливих” ресурсів, можуть спричинити занепад усієї системи. Тому, для запобігання таких сценаріїв, у проектованій системі застосовується шаблон клієнтської стійкості «Вимикач» (“Circuit Breaker”). Цей паттерн бере свій початок з електричного вимикача. В електричній системі вимикач слідкує за тим, щоб у мережі не протікала занадто висока напруга, та, у разі її виявлення, розірве звя’зки з компонентами, що розташовані поза ним, перешкоджаючи їх горінню.

Вимикач зі світу програмного забезпечення слідкуватиме за викликами до віддалених сервісів. Якщо виклик займе забагато часу, вимикач втрутиться та

перерве його. В додаток, вимикач стежитиме за всіма викликами до віддаленого ресурсу, та, якщо достатньо викликів не будуть успішними, він перекриє доступ до цього ресурсу, перешкоджаючи майбутнім викликами йти цим шляхом та даючи їм можливість швидко зазнати невдачі. Цей підхід відомий також як fail-fast.

На рисунку 6.5 зображено декілька сценаріїв викликів віддалених ресурсів із застосуванням вимикача.

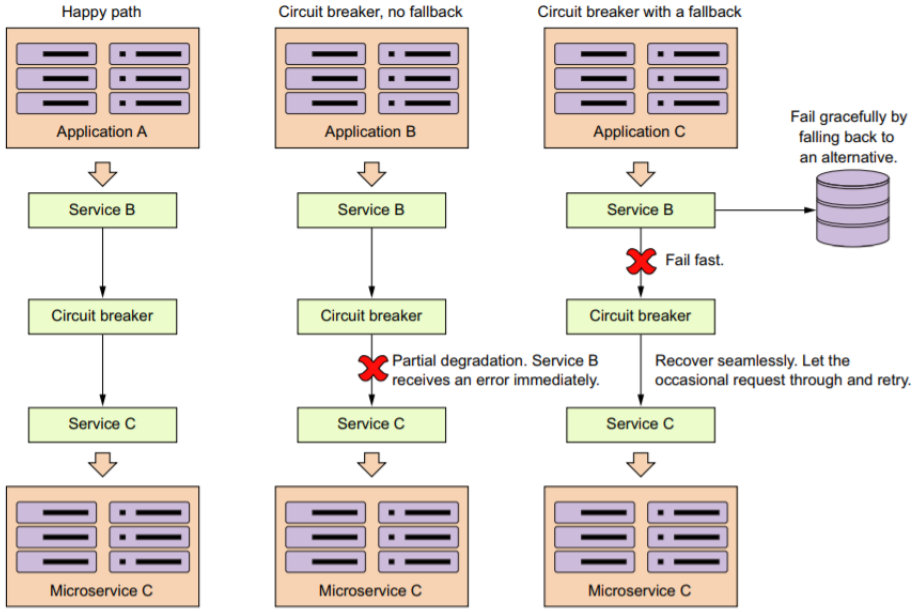

Рисунок 6.5 – Поведінка вимикача у різних станах викликів до віддалених ресурсів

(John Carnell – “Spring Microservices in action”)

У першому випадку, вимикач запускає таймер, і, якщо виклик сервісу B до сервісу C завершиться до того, як сплине час, ніяких проблем не виникне та сервіс B може продовжувати свою роботу.

У другому сценарії, виклик до сервісу C займає більше часу, ніж відведено таймером вимикача, вимикач перерве виклик та почне рахувати кількість невдалих запитів.

У третьому випадку, коли до сервісу С через вимикач проходило забагато невдалих викликів, той перекриє можливість майбутнім викликам викликати повільно працюючий сервіс. Таким чином, сервіс В отримує помилку миттєво, та може використати альтернативний шлях для своєї роботи, якщо такий передбачено. Періодично, вимикач сам надсилатиме запити до «хворого» сервісу, даючи йому можливість відновитися, і, у разі відновлення – знову пропускатиме виклики до нього.

6.4 Розподіл викликів до віддалених ресурсів по різним наборам потоків

Кожен Spring Boot мікросервіс – це невеликий моноліт, який все ще внутрішньо використовує легкий сервлет-контейнер Tomcat від компанії Apache. Коли збудований артифакт розгортається у Tomcat, під процес застосунку сервер виділяє певний набір потоків. Якщо цей сервіс потенційно буде взаємодіяти з декількома віддаленими ресурсами, які можуть бути низькопродуктивними та не завжди доступними, виклики до них можуть привести до занепаду інших внутрішніх процесів застосунку. В такому разі було б непогано виділити під ці небезпечні виклики власний набір потоків. Для цього в системі використовується паттерн «Перегородка» (“Bulkhead”). Bulkhead базується на концепції побудови підводних човнів. Їх секції розділені між собою та абсолютно герметичні, тому, у разі затоплення однієї секції, решта човна не постраждає.

Така ж концепція може бути використана із сервісом, який мусить взаємодіяти з багатьма віддаленими сервісами. Використовуючи цей паттерн, можна розподілити виклики до кожного віддаленого ресурсу по індивідуальним наборам потоків та знизити ризик того, що виклик до повільно відповідаючого ресурсу, вплине на продуктивність усієї системи. Ці набори потоків і грають роль перегородок для сервісу. Кожний віддалений ресурс призначений для одного набору потоків, тому у разі гальмування одного, виклики до інших ресурсів будуть ізольовані.

На рисунках 6.6 та 6.7 проілюстровано звичний набір потоків, що виділяється для всього Java-контейнеру, та розділення потоків для специфічних викликів за допомогою бібліотеки Hystrix від компанії Netflix.

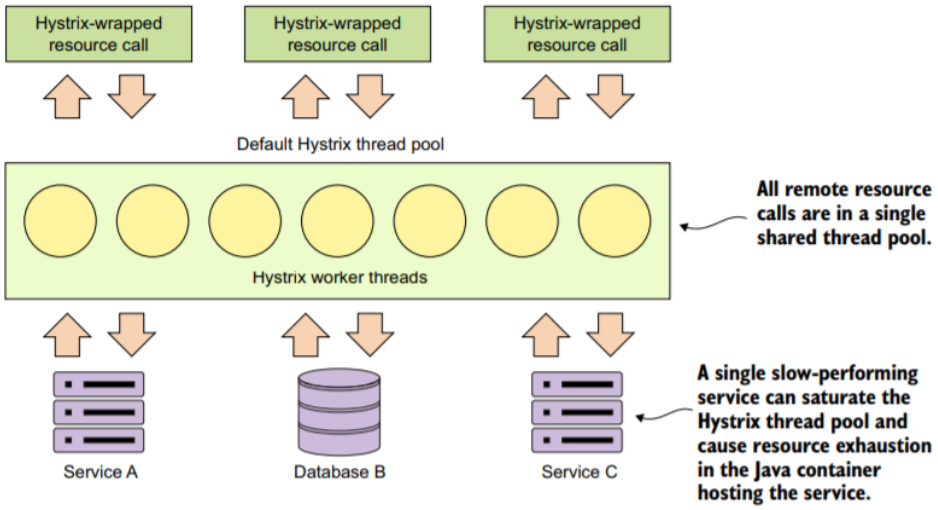

Рисунок 6.6 – Усі виклики до віддалених сервісів використовують єдиний набір потоків

(John Carnell – “Spring Microservices in action”)

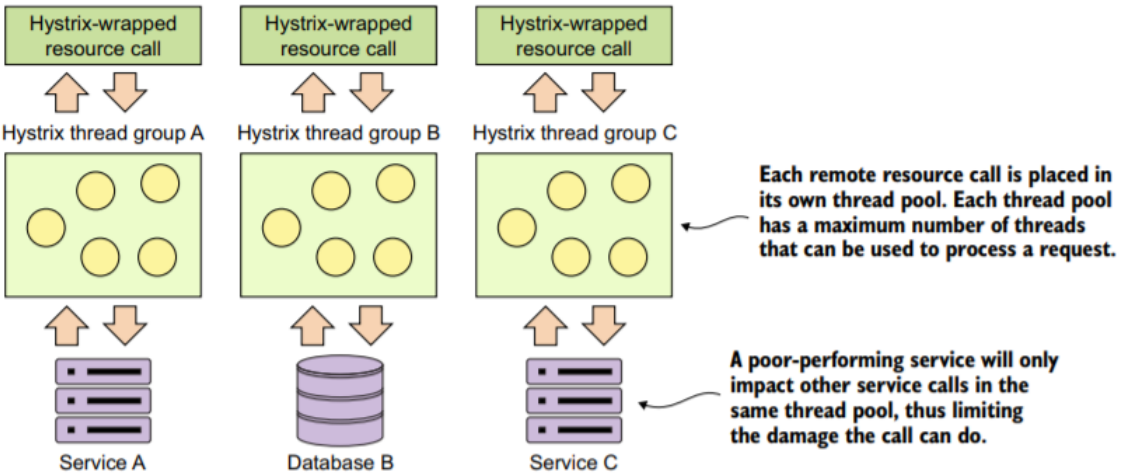

Рисунок 6.7 – Виклики до різних сервісів використовують лише індивідуальний, виділений під них набір потоків

(John Carnell – “Spring Microservices in action”)

6.5 Багатомовне сховище даних

У монолітних та сервіс-орієнтованих застосунках використовувалася та продовжує використовуватися єдина база даних, як правило – SQL-орієнтована. Вона збільшується у розмірах, не відповідає усім можливим бізнес-потребам, стає низькопродуктивною та вразливою.

В розрізі сьогоденних потреб, коли клієнти бажають новий функціонал якомога швидше, системи якомога стійкіше та захист персональних даних – якомога надійніше, єдине сховище вже не може бути нормою.

У розподіленій системі кожен мікросервіс – малий, швидкий, та відповідальний лише за одну задачу, яку він виконує за допомогою тих інструментів, що найкраще до неї пасують. І на сьогоднішньому ринку сховищ даних є інструменти для будь-яких потреб: реляційні, документоорієнтовані, кешовані, колонкоорієнтовані, здатні розгортатися у кластерах та створювати репліки самих себе. Використовуючи для кожного мікросервісу своє сховище даних, розробники не лише правильно підбирають інструменти під відповідну задачу, але ще й роблять усю систему менш вразливою до втрати даних.

Аналізуючи систему, можна зробити висновок, що найкращим рішенням буде використати різні сховища даних для різних потреб. Для стандартизованих даних, таких, як дані про аккаунт користувача, підійде класична реляційна база даних, для нестандартизованих, наприклад, інформації про тренування – документоорієнтована, а інформацію про нагадування потрібно буде дуже часто зчитувати та рідко вставляти, і в цьому випадку гарно виглядає кешоване сховище.

Висновки до розділу №6

Проектуючи систему у мікросервісному стилі, потрібно приймати до уваги дуже багато інфраструктурних проблем та вивчати засоби їх рішення. На щастя, такі вже існують і, більш того, йдуть прямо з коробки фреймворку Spring Cloud. Netflix Eureka за допомогою однієї анотації та одного конфігураційного файлу забезпечить готовий реєстр сервісів. Spring Cloud Config – централізоване, захищене та віддалене сховище налаштувань для кожного із мікросервісів. Netflix Hystrix обгорнить виклики до сторонніх ресурсів у свій вимикач, тому у разі проблем із ними, виклики не будуть довго очікувати на своє завершення. Netflix Zuul залишить лише одну точку доступу до системи, що дозволить захистити бізнес-сервіси, абстрагувати клієнта від них та реалізувати елегантну маршрутизацію.

7 ДЕКОМПОЗИЦІЯ БІЗНЕС-ПРОБЛЕМИ НА МІКРОСЕРВІСИ

Проектуючи систему, можна виділити три основних функції:

- ідентифікація та аутентифікація користувача;
- управління програмою тренувань;
- відправлення нагадувань користувачу різними засобами.

Має сенс виділити три відповідних бізнес-сервіси – Authentication Service, Training Service та Notification Service.

7.1 Бізнес-сервіси

7.1.1 Authentication Service

Сьогодні існує чимало способів захистити певні маршрути застосунку від несанкціонованого доступу. Найпопулярніші з них – протокол OAuth2 та стандарт RFC 7519 для представлення прав користувача між різними сервісами.

JWT (JSON Web Token) реалізує цей стандарт та призначений для надання доступу авторизованим користувачам до інших сервісів без необхідності поширювати свої персональні дані серед інших сервісів. Ці токени можуть бути підписані, використовуючи секретний набір символів (з алгоритмом HMAC), або через публічну чи приватну пару ключів за допомогою алгоритмів шифрування RSA або ECDSA.

JSON Web Token складається з трьох частин: Header, Payload та Signature, розділених символами крапки. Далі ці складові розглядаються більш детально.

Заголовок у більшості випадків складається з двох частин: типу токену (JWT) та алгоритму цифрового підпису (наприклад, HS256). Ці дані у форматі JSON шифруються кодуванням Base64Url та формують першу частину токену.

Друга частина токену – Payload – містить в собі заяви (claims). Заяви – це твердження про сутність (користувача) та додаткові дані. Простіше кажучи, payload може містити в собі унікальне ім’я користувача, його пароль та права.

Payload також зашифрований у Base64Url та формує другу частину JWT-токену.

Третя частина – цифровий підпис. Щоб його створити, знадобиться закодований заголовок, закодований Payload, секрет та алгоритм, що був вказаний у заголовку. Підпис використовується для того, щоб переконатися, що повідомлення не було модифіковано впродовж всього його шляху та, у випадку, якщо токен підписаний приватним ключем, переконатися, що відправник JWT токену дійсно той, за кого себе видає.

На виході ми отримуємо три закодованих у Base64Url рядки, розділених крапками, які можуть бути легко передані у HTML та HTTP середовищах.

Authentication Service відповідає за реєстрацію/логін користувача, а також генерує JWT-токени для запобігання неавторизованого доступу до інших бізнес-сервісів. Відправляючи POST-запит до Authentication Service із логіном та паролем у тілі, користувач, у випадку успішної авторизації, отримає JWT-токен, із яким він вже може робити запити до Training Service та Notification Service для управління програмою тренувань та нагадуваннями відповідно.

Приклад такого токену зображено на рисунку 7.

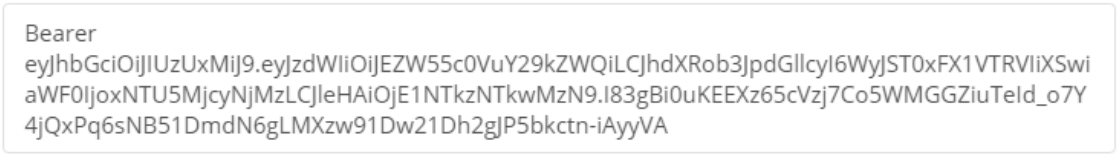

Рисунок 7 – Приклад JSON Web Token

7.1.2 Training Service

Training Service відповідає за складання та управління програмою тренувань. Користувач матиме можливість зробити її максимально гнучкою відповідно до своїх потреб та образу життя, а також відслідковувати власний прогрес набору чи скидування ваги, поступово збільшуючи навантаження.

7.1.3 Notification Service

Notification Service – відповідає за управління різними варіантами повідомлень для користувачів. В системі буде реалізовано три засоби комунікації – електронна пошта, SMS та телефонні дзвінки. Для взаємодії з поштою використовується бібліотека Spring Mail, а для SMS та дзвінків – API американського продукту Twilio.

7.2 Інфраструктурні сервіси

Далі розглядаються Spring Cloud сервіси, що допомагають вирішити інфраструктурні проблеми.

7.2.1 Configuration Server

У розділах вище було згадано багато викликів під час проектування та обслуговування системи, побудованої на базі мікросервісної архітектури. Одним з таких було питання повторюваності сервісів. Іншими словами, розробники та системні адміністратори повинні бути максимально впевнені в тому, що кожного разу, як вони розгортають поруч новий екземпляр сервісу, той буде тієї самої структури та налаштувань, що і його попередники. Отже, потрібне централізоване сховище налаштувань для кожного з сервісів.

Одним з підпроектів фреймворку Spring Cloud є Spring Cloud Config – бібліотека, що дозволяє використовувати сервіс як постачальника налаштувань для інших сервісів, як-то адресу та доступ до бази даних, правила маршрутизації, налаштування захисту, порти, на яких підійматимуться сервіси тощо.

Spring Cloud Config надає розробникам вибір між двома підходами – зберігання налаштувань у файловій системі чи у віддаленому репозиторії. В розроблюваній системі уся чутлива та потенційно вразлива до атак конфігурація зберігатиметься в окремому репозиторії від вихідного коду. При старті Configuration Service зчитає з репозиторію усі конфігураційні файли, а інші сервіси отримуватимуть свої специфічні налаштування через взаємодію з Configuration Service за допомогою REST API.

7.2.2 Discovery Server

Discovery Server – Spring-Boot-застосунок, побудований на базі проекту з відкритим вихідним кодом Netflix Eureka. Споживачі бізнес-сервісу завжди абстраговані від фізичної адреси екземпляру цього бізнес-сервісу. При запуску сервіси зареєструють себе в рєстрі сервісів Eureka. Цей процес повідомить Eureka фізичне знаходження та номер порту кожного екземпляру сервісу. Коли один із сервісів виконуватиме запит до іншого, вбудований балансувальник навантажень Netflix Ribbon звернеться до Eureka щоб дістати інформацію про місцезнаходження екземплярів та закешувати її локально. Періодично, Ribbon знову звертатиметься до Eureka щоб оновити інформацію про місцезнаходження сервісів у себе в кеші.

7.2.3 API Gateway

API Gateway – Spring Boot-застосунок, що використовує бібліотеку Netflix Zuul. По більшій своїй мірі, Zuul грає роль reverse-proxy та реалізує патерн

«єдина точка доступу». Усі запити із зовнішнього світу обов’язкого пройдуть через порт Zuul’a, який керуватиме маршрутами до внутрішніх сервісів.

Окрім цього, на рівні API Gateway можна контролювати доступ до захищених ресурсів, відслідковувати весь життєвий цикл HTTP-запиту, збирати логи та метрики щоб проводити моніторинг стану усієї системи.

Висновки до розділу №7

Проектуючи мікросервісну систему, дуже важливо виділити межі відповідальності для кожного із компонентів. Це потрібно для того, щоб сервіси залишалися малими, легкими для зрозуміння, розробки, підтримки, масштабування та не споживали забагато ресурсів.

В даному розділі бізнес-потреби були успішно декомпозовані у окремі бізнес-мікросервіси, а інфраструктурні проблеми вирішуватимуть інфраструктурні мікросервіси відповідно. Кожен із сервісів залишається малим та із своєю зоною відповідальності, тому можна зробити висновок, що декомпозицію було виконано успішно.

8 ВИБІР СХОВИЩ ДАНИХ

У мікросервісному світі надзвичайно поширеним є підхід Polyglot Persistence. При ньому, система взаємодіє з більше ніж одним сховищем даних, при чому вони, як правило, різних типів. Це дозволяє ізольювати дані користувача від потенційно єдиного вразливого місця атаки. Окрім цього, кожен мікросервіс відповідальний лише за свою конкретну бізнес-задачу та використовує той тип сховища, який найкраще до неї пасує.

8.1 SQL-сховища

Authentication Service відповідальний за дані про аккаунти користувачів. Ця сутність буде статичною, із визначеними полями, тому класична SQL база даних добре впорається із нею.

PostgreSQL – безкоштовна, перевірена, стабільна система управління базами данних із простим SQL-діалектом. Саме тому для цього сервісу було обрано її.

8.2 NoSQL-сховища

У Training Service програма тренувань та їх періодичність можуть бути надзвичайно різноманітними, тому стандартизована SQL-база даних тут не підійде. Натомість у системі використовуватиметься популярна документоорієнтована NoSQL база даних MongoDB, що зберігатиме дані у гнучкому та зручному для клієнтських застосунків форматі JSON, або JavaScript Object Notation.

8.3 Кешовані сховища

У Notification Service зберігатиметься інформація про нагадування. Не стільки важливо, яким способом її зберігати, скільки інтенсивість операцій, що проводитимуться із цими об’єктами. Потрібно буде надзвичайно часто їх вичитувати, часто видаляти та доволі рідко вставляти. В цьому випадку ідеальним вибором буде Redis Cache – легке та швидке сховище даних вигляду «ключ-значення».

Висновки до розділу №8

Багатомовне сховище даних – підхід, що створений саме для високонавантажених розподілених систем. Тому найрозумнішим рішенням буде під кожний мікросервіс виділити окреме сховище даних, яке найкраще пасуватиме для його бізнес-задачі. Для стандартизованих даних підійде SQL-сховище, для нестандартизованих – NoSQL, а для частого зчитування – кешоване.

9 REST API ЯК ЗАСІБ МІЖСЕРВІСНОЇ КОМУНІКАЦІЇ

Усі сервіси, як бізнес-орієнтовані, так і інфраструктурні, спілкуватимуться між собою за допомогою REST (Representational State Transfer) API за синхронним протоколом HTTP. Принцип REST прийшов на зміну SOAP – нелегкому для сприняття протоколу міжсервісної комунікації, який було непросто реалізувати та який передавав дані виключно у форматі XML. SOAP доволі старий, стандартизований та захищений, на відміну від REST. Тому йому й по сьогодні віддають перевагу при проектуванні банківських та інших платіжних систем. REST пропонує не ускладнювати розробку, а використовувати те, що вже є «з коробки» – протокол HTTP, його методи GET, POST, PUT (іноді PATCH), DELETE, та велику кількість статус кодів для обробки відповідей.

В ідеології REST будь-що є ресурс: числа, рядки, об’єкти, документи, медіа файли, зображення та багато іншого. Відповідно, REST може працювати із безліччю форматами даних. Проте найпоширенішим на сьогодні залишається JSON – через його легкість, читабельність, та рідний формат по відношенню до клієнтських застосунків, написаних за домогою мови програмування JavaScript та фреймворків, що на ній базуються.

Метод GET використовується для того, щоб отримати якийсь ресурс, POST – створити, PUT/PATCH – оновити, а DELETE – видалити. Важливо зазначити різницю між методами POST та PUT – на відміну від створення ресурсу, операція оновлення мусить бути ідемпотентною, тобто, надавати один і той самий результат в незалежності від того, скільки разів ця операція була виконана при одних і тих же умовах.

Приклад найпростішого REST API на ресурсі “users” (користувачі):

- GET /users – отримати коллекцію усіх користувачів;
- GET /users/{id} – отримати єдиного користувача за вказаним id;
- POST /users – створити нового користувача;
- PUT /users/{id} – оновити користувача із вказаним id;

– DELETE /users – видалити усіх користувачів;
– DELETE /users/{id} – видалити користувача за вказаним id.

REST API – легкий, інтуїтивно зрозумілий, технологічно нейтральний підхід із можливістю обмінюватися даними рідним для клієнстських застосунків форматом JSON, який використовує лише протокол HTTP і ніяких сторонніх фреймворків чи засобів. Саме тому для більшості веб-додатків, в тому числі і розподілених, цей спосіб коммунікації є найбільш оптимальним та зручним.

Висновки до розділу №9

Через свою легкість, швидкість та близкість до людської мови, REST став де-факто стандартом для міжкомпонентної комунікації. На сьогоднішній день він є найкращим рішенням для дев’яноста відсотків клієнт-серверних застосунків, і для проектованої системи також.

10 СИСТЕМА КОНТЕЙНЕРИЗАЦІЇ

Система контейнеризації – це низькорівневий застосунок, що дозволяє ізолювати будь-який процес операційної системи хоста у так званий «контейнер». В першу чергу це було винайдено для легкої передачі, масштабування та поставки програмних продуктів. До винаходу засобів контейнеризації, системні адміністратори ставили на сервери віртуальні машини над головною операційною системою. І для задачі кожної з цих операційних систем, на кожну з них потрібно було іще зверху встановлювати відповідне програмне запезпечення, бібліотеки, при чому процеси, які запускали у віртуальних машинах використовували ресурси цих віртуальних машин. Кожна віртуальна машина як правило важка, та потребує серйозне залізо під головною операційною системою для продуктивного функціонування. Двигун контейнеризації, зокрема Docker, вирішив цю проблему. Його головна перевага в тому, що контейнер являє собою легкий образ віртуальної машини, в якому працюють лише необхідні процеси і що найважливіше – використовує ресурси головної операційної системи і використовує небагато. Порівняння Docker та віртуальних машин зображено на рисунку 10.1.

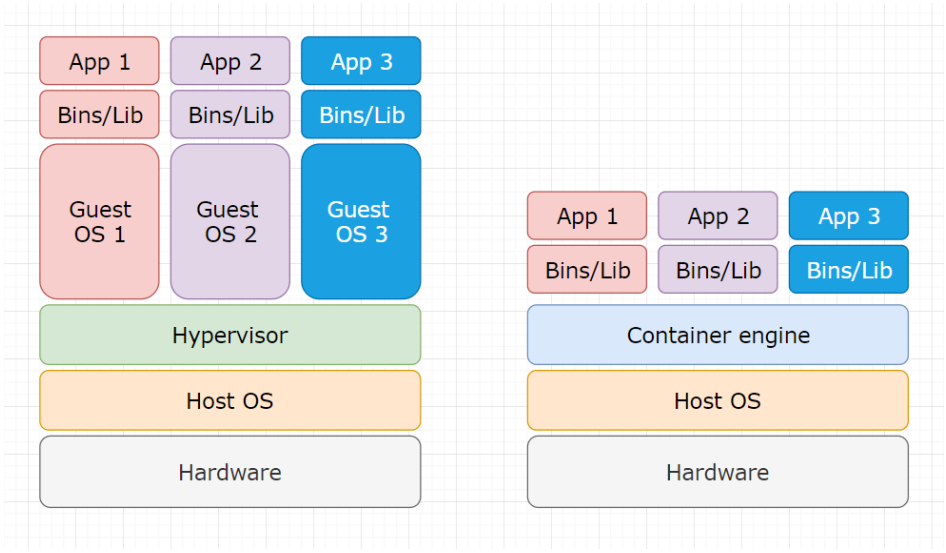

Рисунок 10.1 – Принцип роботи віртуальних машин та Docker

За допомогою простих Dockerfile-скриптів можна запустити будь-який застосунок чи процес в легкому ізольованому оточенні. Це називатиметься контейнером. Контейнери легко передавати іншим користувачам, масштабувати та вони грають дуже важливу роль у процесах неперервної інтеграції та неперервної поставки програмного забезпечення. На рисунку 10.2 зображено найпростіший Dockerfile для запуску в контейнері Notification Service.

Dockerfile

```
1  FROM java:8-jre
2  COPY ./target/notification-service.jar /app/
3  CMD ["java", "-jar", "/app/notification-service.jar"]
4  EXPOSE 8081
```

Рисунок 10.2 – Dockerfile для запуску Notification Service у контейнері

На першому рядку командою FROM вказується образ операційної системи та, опціонально, заздалегідь встановленого на неї програмного забезпечення. В даному випадку неважливо, образ якої операційної системи буде в контейнері, головне – щоб на ній було встановлено Java Runtime Environment версії 1.8. На другому рядку зібраний виконуємий артифакт notification-service.jar копіюється з директорії «target» хостової операційної системи до директорії «app» операційної системи в контейнері. На третьому рядку за допомогою команди CMD виконуються bash-команди, передаючи аргументи як масив. Контейнер прийме команду «java -jar /app/notification-service.jar», яка запустить мікросервіс. На четвертому рядку оголошена команда EXPOSE. Вона зазначає, який порт прослуховуватиме процес у контейнері.

Не можна не зазначити, що файлова система контейнерів повністю ізольована від файлової системи хоста, якщо тільки не налаштувати інше. Це робиться через спеціальні директорії, що називаться volumes. Вони надзвичайно корисні,

коли у контейнері піднімається сховище даних і ці дані треба зберегти навіть після того, як контейнер буде знищено.

Docker став по-справжньому революційною технологією. Відтепер розробники не мусять локально встановлювати важке програме забезпечення. Достатньо лише розгорнути легкий процес у контейнері і працювати з ним.

У комплекті із Docker йде інструмент, що називається Docker Compose. Він призначений для одночасного або послідовного запуску багатьох контейнерів, які можуть залежати один від одного. Розробник або системний адміністратор створює скрипт у форматі yml, в якому прописує налаштування для контейнерів. Фрагмент такого скрипту наведено на рисунку 10.3.

docker-compose.dev.yml

```yaml
services:

  postgres:
    image: postgres:9.6-alpine
    container_name: postgres
    ports:
      - 5432:5432
    volumes:
      - postgres:/data/postgresql
    environment:
      POSTGRES_USER: 'postgres'
      POSTGRES_PASSWORD: 'postgres'
      POSTGRES_DB: 'DoIT-Users'

  mongo:
    image: mongo
    container_name: mongo
    ports:
      - 27017:27017
    volumes:
      - mongo:/data/mongodb
    environment:
      MONGODB_USER: 'mongo'
      MONGODB_PASSWORD: 'mongo'

  redis:
    image: redis:alpine
    container_name: redis
    ports:
      - 6379:6379
    volumes:
      - redis:/data/redis
```

Рисунок 10.3 – Налаштування контейнерів для одночасного старту в скрипті docker-compose.yml

Налаштовувати стартові параметри та взаємодію контейнерів можна дуже гнучко. Фінальне імя контейнера, прокидування внутрішніх портів на хостову операційну систему, налаштування volumes, передача змінних оточення – це мізерна частина з того, що можна робити з контейнерами. Ця технологія швидко завоювала ринок та стала незамінною під час розробки та експлуатації мікросервісів.

Висновки до розділу №10

Коли йде мова про масштабування легких та маленьких застосунків без інвестицій у серверне обладнання, інакше кажучи – горизонтальне масштабування, Docker є найкращим (і єдиним) рішенням. Він дозволяє обгорнути процес кожного мікросервісу чи сховища даних у легкий та швидкий контейнер, який можна дублікувати та контролювати, а docker-compose допомагає налаштувати взаємодію між цими контейнерами. Саме тому в проектованій системі кожен компонент, як бізнес-сервіс, інфраструктурний сервіс чи сховище даних будуть розгортатися у Docker-контейнері.

11 РОЗРОБКА ОКРЕМИХ КОМПОНЕНТІВ

11.1 Discovery Server

Spring Cloud, як і усі інші продукти компанії Pivotal, покликаний максимально полегшити життя для Java-розробників. Завдяки простоті їх використання, програмісти можуть майже повністю абстрагуватись від інфраструктурних проблем, передаючи їх на вирішення фреймворку, та замість цього зосередитись на реалізації бізнес-логіки.

Розроблений компанією Netflix, механізм реєстрації сервісів Eureka, що входить до Spring Cloud, не є вийнятком. Для того, щоб отримати готовий бойовий реєстр сервісів, необхідно зробити 4 елементарних кроки:

1) створити новий Spring Boot додаток;
2) додати залежність spring-cloud-starter-netflix-eureka-server до дескриптора pom.xml;
3) позначити головний клас анотацією @EnableEurekaServer;
4) додати просту конфігурацію до файлу bootstrap.yml у директорії resources.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Рисунок 11.1– Залежність для Eureka у дескрипторі pom.xml

```
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServerApplication {

    public static void main(String[] args) { SpringApplication.run(DiscoveryServerApplication.class, args); }

}
```

Рисунок 11.2 – Головний клас застосунку DiscoveryServerApplication.java позначено анотацією @EnableEurekaServer

На рисунку 11.3 зображено конфігураційний файл для Discovery Service – bootstrap.yml. Далі розглянуто його вміст детальніше.

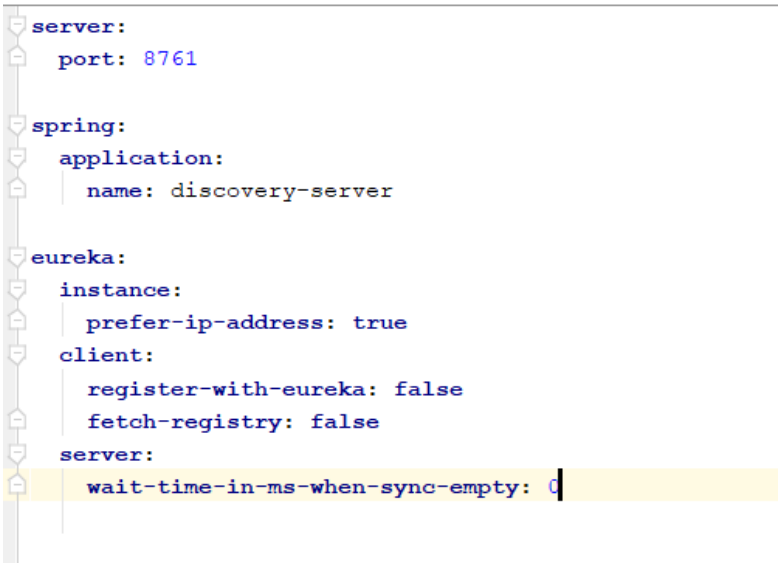

Рисунок 11.3 – Конфігураційний файл bootstrap.yml для Discovery Service

Властивість server.port вказує порт, на якому підніметься застосунок. Властивість spring.application.name задає зручне найменування для сервісу. Eureka.instance.prefer-ip-address вказує на те, щоб сервіси реєструвалися у Eureka за IP-адресою, а не доменним ім'ям. Eureka.client.register-with-eureka – оскільки цей сервіс і є Eureka, то не потрібно реєструвати самого себе. Eureka.client.fetch-registry говорить сервісу, що не треба локально кешувати інформацію про реєстрацію. Eureka.server.wait-time-in-ms-when-sync-empty – час на очікування перед тим, як сервер почне обробляти запити.

Це все, що необхідно зробити для того, щоб отримати Service Discovery прямо з коробки. Запустити додаток можна командою "mvn spring-boot:run". Щоб перевірити його працездатність, треба перейти за адресою http://localhost:8761 (рис. 11.4).

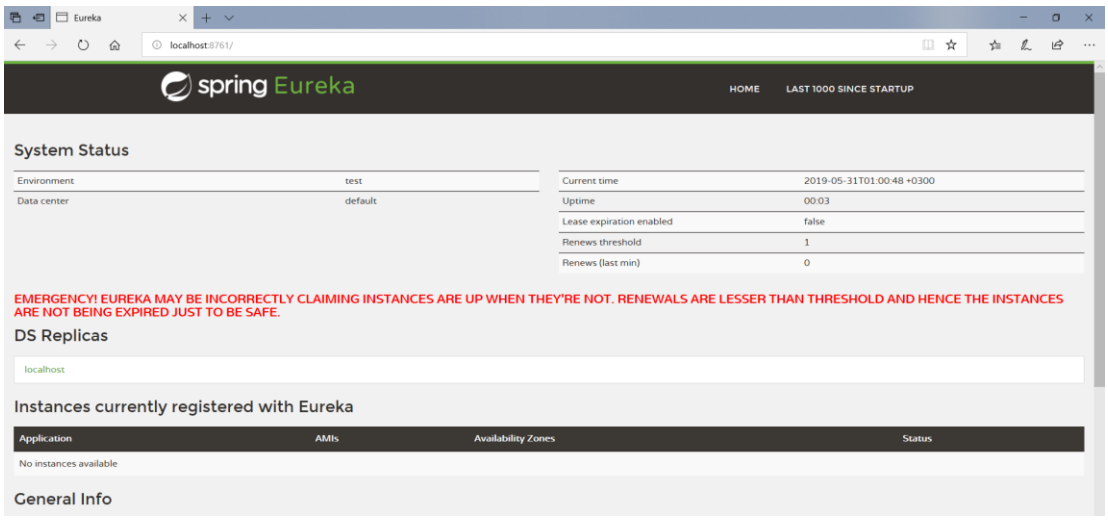

Рисунок 11.4 – Eureka Server успішно запустився та готовий реєструвати сервіси

На рисунку 11.4 у таблиці «Instances currently registered with Eureka» можна побачити повідомлення «No instances available». Це через те, що у Discovery Service поки що не зареєстровано жодного клієнта.

11.2 Configuration Service

Повторюваність – одна з найважливіших умов безпечного розгортання декількох екземплярів сервісу, особливо в хмарних середовищах. Щоб мінімізувати людський фактор впливу на конфігурацію сервісів, а також мати можливість динамічно оновлювати її в залежності від середовища, компанія Pivotal розробила Spring Cloud Config – бібліотеку для централізованого збереження та контролювання конфігурацій мікросервісів. Як і будь-який інший продукт величезного модуля Spring, Spring Cloud Config надзвичайно легко використовувати.

Цей інструмент надає розробникам на вибір два підходи для управління конфігураціями: у файловій системі або на віддаленому репозиторії. В той час як зберігаючи конфігурацію у файловій системі сервера, ми маємо можливість швидко її змінювати, підхід із віддаленим репозиторієм вважається більш захищеним

та канонічним. До того ж, репозиторій можна зробити приватним, а конфігурації зашифрувати. Саме тому був обраний підхід з репозиторієм.

Подібно до налаштування Discovery Server, потрібно виконати чотири простих кроки:

1) створити новий Spring Boot додаток;
2) додати залежність spring-cloud-config-server до дескриптора pom.xml;
3) позначити головний клас анотацією @EnableConfigServer;
4) додати просту конфігурацію до файлу bootstrap.yml у директорії resources.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Рисунок 11.5 – Залежність для Config Server у дескрипторі pom.xml

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

    public static void main(String[] args) { SpringApplication.run(ConfigServerApplication.class, args); }

}
```

Рисунок 11.6 – Головний клас застосунку ConfigServerApplication.java позначено анотацією @EnableConfigServer

Тепер необхідно створити декілька конфігураційних файлів для мікросервісів: api-gateway/api-gateway.yml, auth-service/auth-service.yml, training-service/training-service.yml, notification-service/notification-service.yml та додати їх на віддалений репозиторій. У демонстраційних цілях репозиторій залишено відкритим, а конфігурації – незашифрованими.

Останній крок – додати декілька налаштувань до конфігураційного файлу самого Config Service bootstrap.yml у директорії resources.

Рисунок 11.7 – Конфігураційний файл bootstrap.yml для Config Service

Як було згадано раніше, server.port визначає порт, на якому підніметься застосунок, а spring.application.name – зручне найменування для сервісу. Властивість spring.cloud.config.server.git.uri вказує на те, що використовуватиметься git-орієнтовне сховище та, власне, задається його адреса, а spring.cloud.config.server.git.search-paths – це перелік директорій у репозиторії, в яких мікросервіс шукатиме конфігураційні файли.

Також, окрім конфігурацій, індивідуальних для кожного мікросервісу, в корньовому каталозі репозиторію є файл application.yml. Він зберігає в собі налаштування, спільні для всіх мікросервісів.

Запуск застосунку виконується командою mvn spring-boot:run. На рисунку 11.8 зображено результат виконання GET-запиту за адресою http://localhost:8888/api-gateway/default в програмі Postman.

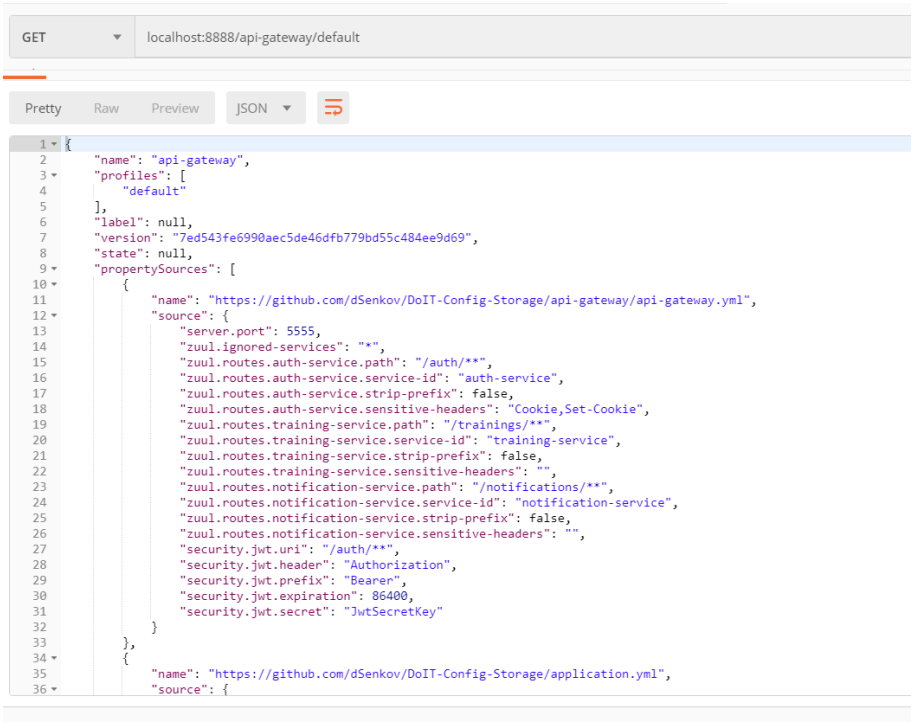

Рисунок 11.8 – Config Service успішно зчитав конфігурацію з віддаленого репозиторію

11.3 API Gateway

Як обговорювалося в попередніх розділах, єдина точка входу – дуже важливий паттерн при проектуванні мікросервісної архітектури. І навіть його Spring Cloud може надати розробникам прямо з коробки. Netflix Zuul – реалізація паттерну API Gateway від компанії Netflix дозволяє не тільки дуже гнучко налаштовувати маршрутизацію, але й реалізовувати клієнтське балансування завдяки його компоненту Ribbon та тісною інтеграцією з Eureka.

Як і минулі сервіси, API Gateway – це Spring Boot застосунок. До дескриптору pom.xml необхідно додати залежності spring-cloud-starter-netflix-zuul для підключення Zuul, spring-cloud-starter-netflix-eureka-client щоб зробити мікросервіс клієнтом в реєстрі сервісів та spring-cloud-config-client для отримання своєї основної конфігурації від Configuration Service (Рис. 11.9).

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-client</artifactId>
</dependency>
```

Рисунок 11.9 – Необхідні залежності в дескрипторі pom.xml

Тепер потрібно додати декілька анотацій у головний клас застосунку. @EnableZuulProxy – для увімкнення API Gateway Zuul. @EnableDiscoveryClient – для того, щоб застосунок при кожному старті реєстрував себе у реєстрі Eureka. @RefreshScope – щоб мати можливість динамічно оновлювати конфігурацію від Configuration Service, коли застосунок вже запущено. Головний клас мікросервісу зображено на рисунку 11.10.

```
@RefreshScope
@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class ApiGatewayApplication {

    public static void main(String[] args) { SpringApplication.run(ApiGatewayApplication.class, args); }

}
```

Рисунок 11.10 – Налаштований головний клас застосунку ApiGatewayApplication.java

У конфігураційному файлі bootstrap.yml необхідно вказати лише адресу Configuration Service за ключем “spring.cloud.config.uri” та

“spring.application.name”. Оскільки API Gateway є клієнтом Discovery Service, то властивість “spring.application.name” буде використана Netflix Eureka як ключ, за яким зберігатиметься коллекція доступних екземплярів сервісу у реєстрі сервісів.

При запуску API Gateway зчитає з Configuration Service свої налаштування та зареєструє себе у Discovery Service. Перевірити набір сервісів, що зареєстровані у Eureka, можна, запустивши додаток командою “mvn spring-boot:run” та перейшовши за адресою http://localhost:8761 (рис. 11.11).

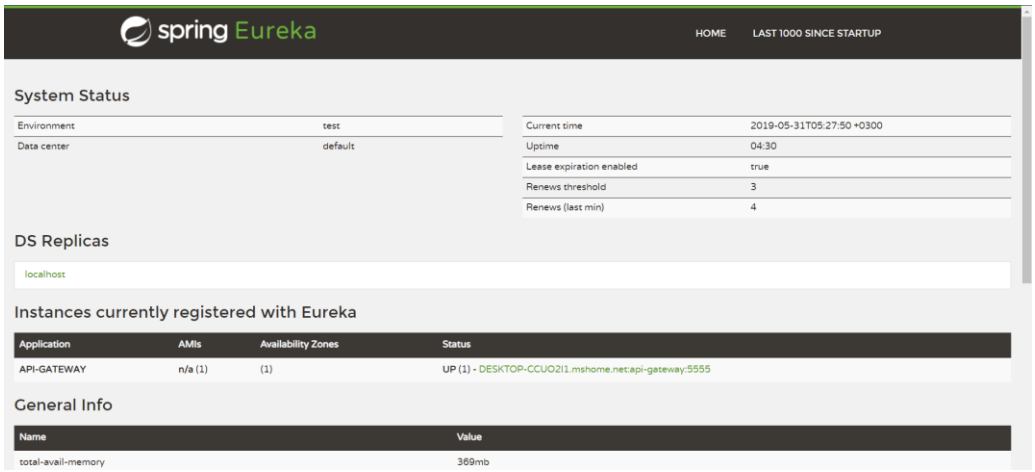

Рисунок 11.11– Сервіс API Gateway успішно зареєстрував себе у Eureka

Якщо відправити GET-запит на http://localhost:5555/actuator/routes, то можна побачити інформацію про діючу маршрутизацію.

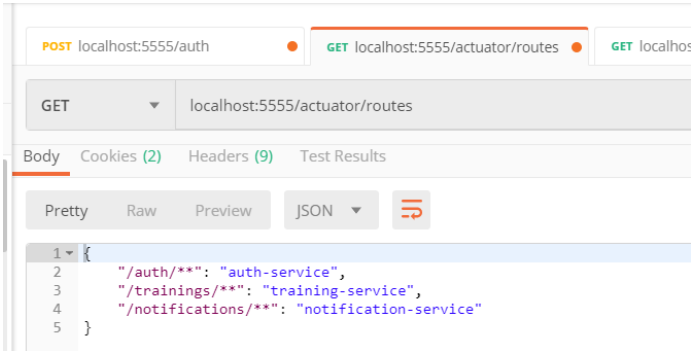

Рисунок 11.12 – Діючі маршрути та сервіси, що за них відповідають

З рисунку 11.12 зрозуміло, що за всі маршрути, що мають префікс “/auth” відповідальний Authentication Service, за “/trainings” – Training Service, за “/notifications” – Notification Service. Маршрути Training Service та Notification Service дуже бажано захистити, щоб лише авторизовані користувачі могли ними користуватися.

Для того, щоб користувач не надсилав свої чутливі дані кожного разу, як хоче скористатися захищеним сервісом, розроблено механізм видачі токенів за логіном та паролем у Authentication Service далі, а на рівні Gateway встановлено фільтр, який перевірятиме, чи знаходиться всередині запита у заголовку “Authorization” JWT-токен, та, якщо знаходиться, чи валідний він.

Фільтр на рівні Gateway представлений у класі JwtTokenAuthenticationFilter.java. Лістинг цього класу наведено у додатку А1.

11.4 Authentication Service

Authentication Service – це також Spring Boot застосунок, що має залежності на Configuration Service та Discovery Service, тому, при запуску, він зчитає свою конфігурацію та зареєструє себе в реєстрі сервісів.

Цей мікросервіс відповідає за роботу з сутністю користувача, його авторизацію та аутентифікацію. Оскільки користувач – це сутність з визначеними та стандартизованими полями, для збереження даних про нього непогано буде використати SQL-базу, наприклад, PostgreSQL.

Насправді, кожен бізнес-мікросервіс – це невеликий моноліт, із тими самими трьома рівнями – рівнем відображення або презентації, рівнем бізнес-логіки та рівнем доступу до даних. Authentication Service не вийняток, він також

складається з цих трьох шарів. Застосунок має класс-контролер, що обробляє вхідні HTTP-запити, сервіс, що є посередником між шаром презентації та доступу до даних а також кодує та декодує паролі користувачів, та шар доступу до бази даних, що реалізується за допомогою фреймворку Spring Data.

Spring Data – це вишукане напрацювання зверху над найпопулярнішим Object-Relation Mapping (ORM) фреймворком серед Java-розробників Hibernate. Для реалізації шару доступу до даних, треба лише оголосити інтерфейс, що наслідуватиметься від CrudRepository<T, ID> або JpaRepository<T, ID> або PagingAndSortingRepository<T, ID>, де T – узагальнений тип користувача, а ID – узагальнений тип його унікального ідентифікатора. Лише наслідуючи один з цих інтерфейсів, можна одразу отримати базові методи для Create-Read-Update-Delete операцій. Інтерфейс, оголошений розробником, не потрібно реалізовувати – під час виконання програми за допомогою метаданих Spring Data самостійно створить його реалізацію. Окрім цього, Spring Data здатна самостійно створювати методи, що виконують запити до даних, базуючись лише на назві методу. Це надзвичайно потужний механізм, якшо тільки, звісно, розробники дотримуються певних обмежень у сигнатурах цих методів.

Друга функція цього сервісу – це генерація JWT-токенів при відправленні POST-запиту із даними про користувача в тілі запиту на маршрут “/auth”. Користувач має право отримати такий токен та користуватися іншими захищеними сервісами тоді й лише тоді, коли він авторизувався в системі. Фільтр, що генерує JWT-токени, представлений у класі JwtUsernameAndPasswordAuthenticationFilter.java. Лістинг цього класу наведено в додатку А2.

11.5 Training Service

Training Service – це також Spring Boot застосунок, що має залежності на Configuration Service та Discovery Service, тому, при запуску, він зчитає свою конфігурацію та зареєструє себе в реєстрі сервісів.

Training Service призначений для простого управління програмою тренувань користувача – створення, оновлення, додавання нових тренувань, вправ, редагування дню та часу тренувань, редагування вправ тощо.

Оскільки програма тренувань та вправи в ній повинні бути максимально гнучкими, то SQL-подібне сховище даних тут не підійде. Натомість сюди дуже пасуватиме документоорієнтована NoSQL база, наприклад, MongoDB, яка не змушує коллекції об’єктів бути жорсткостандартизованими та зберігає дані у рідному для JavaScript-додатків форматі JSON.

Training Service – також типовий маленький моноліт, із трьома шарами: презентації, бізнес-логіки та доступу до даних. Рівень презентації представлений класом TrainingController.java, який обробляє HTTP-запити та обмінюється даними з клієнтом у форматі JSON. Рівень бізнес-логіки представлений класом MongoTrainingService.java, а рівень доступу до даних – інтерфейсом TrainingRepository.java. Він наслідує інтерфейс MongoRepository<T, ID>, який у внутрішній реалізації має усе необхідне для взаємодії з MongoDB.

MongoDB була обрана не випадково. Говорячи про доменну модель цього сервісу, якшо сутність Training можна якось стандартизувати, то сутність Exercise ніяк не можна, вона повинна бути максимально гнучкою, будь то силова вправа, чи кардіо чи будь-який інший тип.

11.6 Notification Service

Notification Service – це також Spring Boot застосунок, що має залежності на Configuration Service та Discovery Service, тому, при запуску, він зчитає свою конфігурацію та зареєструє себе в реєстрі сервісів.

Цей мікросервіс відповідальний за відправлення користувачам різних нагадувань – лист на електронну пошту, SMS-повідомлення чи телефонний дзвінок. Для надсилання листів використовується бібліотека Spring Mail, а для SMS та дзвінків – API сервісу Twilio. Twilio – це продукт, що надає послуги відео- та аудіо-коммунікації програмним способом.

Користувачів може бути багато, а запитів на нагадування ще більше. Об’єкт нагадування не потрібно зберігати у сховищи якомога довше, він там потрібен лише до відпрацювання. Саме тому було обрано Redis Cache у якості сховища даних для нагадувань.

Користувачів може бути багато, а нагадувань ще більше. Окрім того, багато користувачів можуть створювати нагадування на одну й ту ж дату, один й той же час. Через такі умови були прийняті наступні дизайнерські рішення.

Коли на POST-запит прийде нагадування, в ньому знаходитиметься його тип, адреса електронної пошти чи номер мобільного, з якими треба взаємодіяти, унікальний індентифікатор користувача, а також дата та час до хвилин, коли це нагадування треба реалізувати. Хеш Redis’у із назвою «NOTIFICATIONS» (аналог до таблиці у SQL-сховищі та коллекції у NoSQL-сховищі) матиме в якості ключа об’єкт класу LocalDateTime з пакету java.time, а в якості значення – коллекцію об’єктів типу Notification для різних користувачів для цієї хвилини цієї години цієї дати. Notification – це абстрактний клас, який має три реалізації. Діаграма класів для Notification Service представлена в додатку В.

У мікросервісі є клас ExecutorNotificationService.java, який періодично виконуватиме запланований метод – checkForScheduledNotifications(). Цей метод щохвилини братиме в якості ключа поточну дату та час до хвилин, та зробить запит в кеш щоб з'ясувати, чи не заплановано на цей момент якихось нагадувань. Якщо ж коллекція, яка йому повернеться, не буде порожньою, то кожне нагадування негайно почне виконуватися, а коли виконається – його можна видалити, як і всі нагадування цього цього моменту часу.

Оскільки зчитування зі сховища відбувається щохвилини, а вставка в нього чи видалення – набагато рідше, ще раз можна переконатися, що легкий Redis кеш – найкраще рішення для даної задачі.

Висновки до розділу №11

В цьому розділі за допомогою Java-фреймворку Spring Cloud було успішно та легко розроблено інфраструктурні сервіси та описано базові концепції для розробки бізнес-сервісів. API Gateway було побудовано використовуючи Netflix Zuul, Discovery Server за допомогою Netflix Eureka, Configuration Server із використанням Spring Cloud Config. Кожен із сервісів являє собою Spring-Boot застосунок, що всередині має легкий сервлет-контейнер Apache Tomcat для передачі даних по протоколу HTTP. Authentication Service видає аутентифікованим користувачам JWT-токени та взаємодіє із реляційною СУБД PostgreSQL. Notification Service відповідає за гнучку програму тренувань та зберігає і вичитує її із NoSQL-сховища MongoDB. Notification Service відповідає за створення, видалення та обробку нагадувань, зберігаючи їх у Redis Cache та відправляє запити до сервісу програмованої телефонної комунікації Twilio.

12 АВТОМАТИЗОВАНЕ РОЗГОРТАННЯ СИСТЕМИ ЗА ДОПОМОГОЮ DOCKER-COMPOSE

Коли система збільшується за кількістю компонентів, кожен із них мусить бути розгорнутий окремо. Більш того, розгортати їх необхідно у правильній послідовності та із правильними налаштуваннями. Людський фактор у такій ситуацій вкрай великий. Проте, у комплекті із Docker є інструмент Docker Compose, що дозволяє за допомогою одного файлу та однієї команди розгорнути усі компоненти системи у вигляді Docker-контейнерів одночасно.

При описі кожного сервісу потрібно вказати або Docker-image, на базі котрого треба створити контейнер за допомогою директиви image, або шлях до Dockerfile, з якого необхідно його зібрати директивою context (та, за бажанням, dockerfile). Також можна вказувати фінальне ім’я для контейнеру директивою container_name, прописувати налаштування для відповідності внутрішнього порту контейнера зовнішньому директивою ports, вказувати змінні оточення директивою environment, вказувати залежності від інших контейнерів директивою depends_on та багато іншого. Також його можна використовувати не тільки при локальному запуску, але й у хмарних сервісах.

Щоб запустити усі сервіси, описані у скрипті одночасно, потрібно перейти до директорії із файлом та виконати команду docker-compose up якщо файл називається docker-compose.yml, або вказати додатково назву файлу, якщо вона відрізняться, наприклад docker-compose -f docker-compose.dev.yml up. Вміст файлу docker-compose.dev.yml можна знайти у додатку Б.1.

Висновки до розділу №12

У цьому розділі було описано як легко та інтуїтивно зрозуміло забезпечити автоматизоване розгортання усіх компонентів системи та описати залежності між ними за допомогою інструменту docker-compose.

ВИСНОВКИ

У даному дипломному проекті була спроектована система для контролю здорового способу життя із можливістю гнучко складати програму тренувань та встановлювати нагадування зручними способами, що не залежать від стану застосунку, а саме лист на електронну пошту, SMS-повідомлення чи телефонний дзвінок. Мікросервісний підхід до проектування архітектури серверної частини зробив систему стійкою, захищеною, легкомасштабованою та легкопідтримуємою.

Фреймворк Spring Cloud допоміг легко вирішити багато інфраструктурних питань розподіленої системи. Netflix Eureka за допомогою однієї анотації та одного конфігураційного файлу забезпечила готовий реєстр сервісів. Spring Cloud Config забезпечив централізоване, захищене та віддалене сховище налаштувань для кожного із мікросервісів. Netflix Hystrix обгорнув виклики до комунікаційного API Twilio у свій вимикач, тому у разі проблем із хмарним сервісом, виклики не будуть довго очікувати на своє завершення. Netflix Zuul залишил лише одну точку доступу до системи, що дозволило захистити бізнес-сервіси, абстрагувати клієнта від них та реалізувати елегантну маршрутизацію. Docker дозволив запускати процеси мікросервісів у ізольованому оточенні, що дозволить DevOps-інженерам легко масштабувати систему горизонтально, а також розгортати усі компоненти автоматизовано лише однією командою.

В цілому, проект був успішно виконаний, система спроектована з урахуванням усіх потреб користувачів та вимог до сучасних клієнт-серверних застосунків.

У майбутньому планується запровадження процесу неперервної інтеграції за допомогою систем Jenkins або TravisCI, та неперервної поставки із розгортанням у хмарному середовищі Amazon Web Services. І, звісно, реалізація Andoid та IOS-застосунків для взаємодії із серверною частиною.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Spring Microservices in Action / John Carnell / New-York: «Manning Publications Co.» / 2017
2. .NET Microservices: Architecture for Containerized .NET Applications – Cesar de la Torre, Bill Wagner, Mike Rousos / Washington: «Microsoft Developer Division» [Електронний ресурс]: Режим доступу: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/
3. Микросервисная архитектура, Spring Cloud и Docker / sqshq [Електронний ресурс]: Режим доступу: https://habr.com/ru/post/280786/
4. A Guide to Cron Expressions / Baeldung [Електронний ресурс]: Режим доступу: https://www.baeldung.com/cron-expressions
5. Negotium-task-manager-vol.2 / Igor Dmitriev [Електронний ресурс]: Режим доступу: https://github.com/igor-dmitriev/negotium-task-manager-vol2
6. Microservices with Spring Boot – Authentication with JWT (Part 3) / Omar Elgabry [Електронний ресурс]: Режим доступу: https://medium.com/omarelgabrys-blog/microservices-with-spring-boot-authentication-with-jwt-part-3-fafc9d7187e8
7. Spring Security Reference / Ben Alex, Luke Taylor, Rob Winch [Електронний ресурс]: Режим доступу: https://docs.spring.io/spring-security/site/docs/3.2.3.RELEASE/reference/htmlsingle/
8. Introduction to JSON Web Tokens [Електронний ресурс]: Режим доступу: https://jwt.io/introduction/
9. Spring-boot microservice with centralized authentication (ZUUL+Eureka+JWT) / Arjun Nagathankandy [Електронний ресурс]: Режим доступу: https://medium.com/@arjunac009/spring-boot-microservice-with-centralized-authentication-zuul-eureka-jwt-5719e05fde29

10. Spring Cloud Security with Netflix Zuul / Bharat Raj Meriyala [Електронний ресурс]: Режим доступу: https://medium.com/@bharatrajmeriyala/spring-cloud-security-with-netflix-zuul-2ef04a1dcfb
11. Security: Spring Boot features [Електронний ресурс]: Режим доступу: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-security.html
12. Introduction to Spring Data MongoDB / Baeldung [Електронний ресурс]: Режим доступу: https://www.baeldung.com/spring-data-mongodb-tutorial
13. Spring Data Redis in Spring Boot Example / TechPrimers [Електронний ресурс]: Режим доступу: https://www.youtube.com/watch?v=eYfopvusG_s
14. Overview of Docker Compose [Електронний ресурс]: Режим доступу: https://docs.docker.com/compose/overview/
15. C# 4.0 / Herbert Schildt / New-York: «The McGraw-Hill» / 2010
16. Дипломний проект бакалавра. Розробка, оформлення, захист / Дорогой Ярослав Юрійович, Дорошенко Катерина Сергіївна, Репнікова Наталья Борисівна, Юрчук Леонід Юрійович / КПІ ім. Ігоря Сікорського / 2018

ДОДАТОК А Лістинг класів, що реалізовують фільтри у HTTP-запитах

А.1 Лістинг класу JwtTokenAuthenticationFilter.java сервісу API Gateway

```
@RequiredArgsConstructor
public class JwtTokenAuthenticationFilter extends OncePerRequestFilter {

    private final JwtConfig jwtConfig;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain chain) throws ServletException, IOException {
        String header = request.getHeader(jwtConfig.getHeader());
        if (header == null || !header.startsWith(jwtConfig.getPrefix())) {
            chain.doFilter(request, response);
            return;
        }
        String token = header.replace(jwtConfig.getPrefix(), "");
        try {
            Claims claims = Jwts.parser()
                    .setSigningKey(jwtConfig.getSecret().getBytes())
                    .parseClaimsJws(token)
                    .getBody();

            String username = claims.getSubject();
            if (username != null) {
                @SuppressWarnings("unchecked")
                List<String> authorities = (List<String>) claims.get("authorities");
                UsernamePasswordAuthenticationToken auth = new
UsernamePasswordAuthenticationToken(username, null,

authorities.stream().map(SimpleGrantedAuthority::new).collect(Collectors.toList()));
                SecurityContextHolder.getContext().setAuthentication(auth);
            }
        } catch (Exception e) {
            SecurityContextHolder.clearContext();
        }
        chain.doFilter(request, response);
    }
}
```

А.2 Лістинг класу JwtUsernameAndPasswordAuthenticationFilter.java сервісу Authentication Service

```
public class JwtUsernameAndPasswordAuthenticationFilter extends
UsernamePasswordAuthenticationFilter {

    private AuthenticationManager authManager;
    private final JwtConfig jwtConfig;

    public JwtUsernameAndPasswordAuthenticationFilter(AuthenticationManager
authManager, JwtConfig jwtConfig) {
        this.authManager = authManager;
        this.jwtConfig = jwtConfig;
        this.setRequiresAuthenticationRequestMatcher(new
AntPathRequestMatcher(jwtConfig.getUri(), "POST"));
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request,
HttpServletResponse response) throws AuthenticationException {
        try {
            UserCredentials creds = new
ObjectMapper().readValue(request.getInputStream(), UserCredentials.class);
            UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(
                    creds.getUsername(), creds.getPassword(), Collections.emptyList());
            return authManager.authenticate(authToken);
        } catch (IOException e){
            throw new RuntimeException(e);
        }
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request,
HttpServletResponse response, FilterChain chain, Authentication auth) throws
IOException, ServletException {
        Long now = System.currentTimeMillis();
        String token = Jwts.builder()
                .setSubject(auth.getName())
                .claim("authorities", auth.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority).collect(Collectors.toList()))
```

Продовження додатку А.2

```
            .setIssuedAt(new Date(now))
            .setExpiration(new Date(now + jwtConfig.getExpiration() * 1000))
            .signWith(SignatureAlgorithm.HS512, jwtConfig.getSecret().getBytes())
            .compact();
        response.addHeader(jwtConfig.getHeader(),
            jwtConfig.getPrefix().replace(" ", "") + " " + token);
    }
}
```

ДОДАТОК Б Допоміжні скрипти

Б.1 Файл docker-compose.dev.yml для локального розгортання системи

```yaml
version: '3.7'

services:

  postgres:
    image: postgres:9.6-alpine
    container_name: postgres
    ports:
      - 5432:5432
    volumes:
      - postgres:/data/postgresql
    environment:
      POSTGRES_USER: 'postgres'
      POSTGRES_PASSWORD: 'postgres'
      POSTGRES_DB: 'DoIT-Users'

  mongo:
    image: mongo
    container_name: mongo
    ports:
      - 27017:27017
    volumes:
      - mongo:/data/mongodb
    environment:
      MONGODB_USER: 'mongo'
      MONGODB_PASSWORD: 'mongo'

  redis:
    image: redis:alpine
    container_name: redis
    ports:
      - 6379:6379
    volumes:
      - redis:/data/redis

  config-server:
    build:
      context: config-server
      dockerfile: Dockerfile
    container_name: config-server
    ports:
      - 8888:8888

  discovery-server:
    build:
      context: discovery-server
      dockerfile: Dockerfile
    container_name: discovery-server
    ports:
      - 8761:8761
```

Продовження додатку Б.1

```
  api-gateway:
    build:
      context: api-gateway
      dockerfile: Dockerfile
    container_name: api-gateway
    ports:
      - 5555:5555
    depends_on:
      - config-server
      - discovery-server

  auth-service:
    build:
      context: auth-service
      dockerfile: Dockerfile
    container_name: auth-service
    ports:
      - 8080:8080
    depends_on:
      - config-server
      - discovery-server
      - postgres
      - api-gateway
  notification-service:
    build:
      context: notification-service
      dockerfile: Dockerfile
    container_name: notification-service
    ports:
      - 8081:8081
    depends_on:
      - config-server
      - discovery-server
      - redis
      - api-gateway
  training-service:
    build:
      context: training-service
      dockerfile: Dockerfile
    container_name: training-service
    ports:
      - 8082:8082
    depends_on:
      - config-server
      - discovery-server
      - mongo
      - api-gateway

volumes:
  postgres:
  mongo:
  redis:
```